SOLVING POINT-REACTOR KINETICS EQUATIONS USING EXPONENTIAL MOMENT METHODS

THESIS

Paul M. Thelen, Civilian

AFIT-ENP-13-M-34

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENP-13-M-34

SOLVING POINT-REACTOR KINETICS EQUATIONS USING EXPONENTIAL MOMENT METHODS

THESIS

Presented to the Faculty

Department of Engineering Physics

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Nuclear Engineering

Paul M. Thelen, BS

Civilian

March 2013

AFIT-ENP-13-M-34

SOLVING POINT-REACTOR KINETICS EQUATIONS USING EXPONENTIAL MOMENT METHODS

Paul M. Thelen, BS

Civilian

Approved:

_____	_____
Kirk Mathews, PhD (Chairman)	Date
_____	_____
John McClory, PhD (Member)	Date
_____	_____
LTC Stephen McHale (Member)	Date

AFIT-ENP-12-M-34

ABSTRACT

A robust method of solving the reactor point kinetic equations was designed using exponential moment methods. Although the method requires a relatively large number of calculations to complete, the accuracy ensured by each individual step calculation allows larger time steps to be used. The algorithm designed was verified to converge to the correct value as step size was reduced. Additionally, the algorithm can take steps much larger than the average neutron lifetime while maintaining some precision. An error control scheme was designed based on changes observed in the results as a function of time step size. The error control adaptively approaches optimal step sizes within a factor of two for given tolerances. When used in conjunction with our algorithm, most cases show large mitigation of computational cost.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. Kirk Mathews. The discoveries made within this thesis are only a fraction of the discussions and observations we shared. Of the experience I gained during my endeavor, the most important was not knowledge of reactors, nuclear engineering or computer science, but that of problem solving and discipline; all of which fall under Dr. Mathews' expertise.

Paul M. Thelen

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

SOLVING POINT-REACTOR KINETICS EQUATIONS USING EXPONENTIAL MOMENT METHODS

I. INTRODUCTION

I.A. Background

One of the main properties of interest within a nuclear reactor is the neutron population at any given time. Mathematically, this is denoted $n(t)$, the number of neutrons or more specifically, the *neutron density* at time t. This neutron density, along with several other properties of the reactor will determine the behavior of the reactor. Each fission within the reactor will produce more neutrons, each one with a probability of either creating a fission or being lost to some loss mechanism. Provded this information, one can map $n(t)$. Each fission consumes one neutron and produces an average number of neutrons, ν, creating the potential of multiplying the neutron population. When the probability of all the various loss mechanisms are taken into account, one can find the *effective multiplication factor* of neutrons for each generation, which is denoted k_{eff}. This multiplication factor is then used to determine the relative increase between generations of fission, also known as *reactivity*, with the symbol ρ. The two are related by

$$\rho = \frac{k_{eff} - 1}{k_{eff}} \quad . \tag{1}$$

Neutrons quickly go through the motion of birth, absorption and fission and most reactions quickly end. The time spent travelling and scattering in a reactor until the neutron is absorbed is known as the reproduction lifetime. The *average reproduction lifetime* is denoted with the symbol Λ.

Reactors are carefully designed to control the neutron population using *delayed neutrons*. Neutrons produced through fission belong to one of two different categories. Prompt neutrons occur directly from the fission process and quickly go through the cycle. However, some of the fission fragments left from the fission will not initially produce a neutron. Instead they will decay through other mechanisms, mainly *beta decay*. Some of these decay chains eventually lead to a decay that will emit a neutron, known as a delayed neutron. The fraction of the neutrons from fission that are delayed is the *delayed neutron fraction* and has the symbol β. If the reactivity is less than 0, the reactor is said to be *subcritical*. A reactor with a reactivity exactly equal to 0 is *critical*. Once the reactivity exceeds 0 the reactor is *supercritical*. Finally, if the number of prompt neutrons produced alone can exceed the neutron population, a special case of super criticality occurs known as prompt critical, or prompt supercritical. This will result in a rapid increase in the neutron population on the time scale of

the neutron lifetime. Generally speaking, reactors are not designed to go prompt critical because the reactor would be difficult to control in this state.

Neutrons can be supplementally added to the reactor in through means other than fission. Often a plutonium-beryllium neutron source is placed near a reactor and produces extra neutrons at a near constant rate through an alpha, neutron reaction. The reactor materials often produce neutrons as well through spontaneous fission. These additional neutrons that are added to the reactor through means that are not influenced by the current neutron population are denoted as $S(t)$, or the *average source rate density*.

Temperature, materials, geometry and current conditions of the reactor will determine the functional form of $S(t)$ and $\rho(t)$. Reactor kinetics is the study of time-dependent phenomena including the use of these conditions in order to determine the neutron density $n(t)$. The system of differential equations that motivates this work is the point reactor kinetic equations (PRKEs). The PRKEs are a system of nonhomogeneous differential equations of the following form:

$$\frac{dn(t)}{dt} = \left(\frac{\rho(t) - \beta}{\Lambda}\right) n(t) + \sum_i \lambda_i c_i(t) + S(t) \tag{2}$$

$$\frac{dc_i(t)}{dt} + \lambda_i c_i(t) = \frac{\beta_i}{\Lambda} n(t) \tag{3}$$

where $n(t)$ is the neutron density at time t; $c_i(t)$ is the precursor density of group i at time t; $\rho(t)$ is the reactivity at time t; $S(t)$ is the source rate density

at time t; β_i is the delayed neutron fraction for group i; β is the sum of all delayed neutron fraction groups; Λ is the neutron lifetime; and λ_i is the decay constant for group i. In matrix form these equations are

$$\begin{bmatrix} n'(t) \\ c_1{}'(t) \\ c_2{}'(t) \\ . \\ . \\ . \\ c_n{}'(t) \end{bmatrix} = \begin{bmatrix} \left(\frac{\rho(t)-\beta}{\Lambda}\right) & \lambda_1 & \lambda_2 & . & . & . & \lambda_n \\ \left(\frac{\beta_1}{\Lambda}\right) & -\lambda_1 & 0 & . & . & . & 0 \\ \left(\frac{\beta_2}{\Lambda}\right) & 0 & -\lambda_2 & 0 & . & . & 0 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ \left(\frac{\beta_n}{\Lambda}\right) & 0 & . & . & . & 0 & -\lambda_n \end{bmatrix} \begin{bmatrix} n(t) \\ c_1(t) \\ c_2(t) \\ . \\ . \\ . \\ c_n(t) \end{bmatrix} + \begin{bmatrix} S(t) \\ 0 \\ 0 \\ . \\ . \\ . \\ 0 \end{bmatrix} \tag{4}$$

.

I.B. Motivation

Solving the differential equations that compose the reactor kinetics problem has not been a very difficult task for modern mathematics packages. Depending on the form of $\rho(t)$ and $S(t)$, the solution might even be closed form. Common iterative methods used by state of the art mathematics packages today include trapezoid rule, extrapolation techniques or some form of Runge-Kutta methods. The main issue with the reactor kinetics equations (RKEs) is the stiffness of the problem. Fission in the reactor leads to the creation of many daughter nuclides that are precursors to delayed neutrons. The amount of time it takes for these

various precursors to decay is different by orders of magnitude. Additionally, all of the precursors work in a timescale that is much longer than prompt lifetime. In order to accurately estimate the conditions of the reactor at any given time, some of these methods require very small step sizes to keep track of the short time scale behaviors that can build up over a long time period of interest. A few choices are available to use. Multistep methods are the standard for solving the PRKEs, but the complexity of other RKEs makes multistep methods impractical. Simple explicit methods, such as Euler, require many small steps because of first order convergence. Larger steps can improve computational cost but ruin the accuracy of an answer. More complex implicit techniques may increase the order of convergence but also increase computational cost. This dilemma is really a question of the cost of computation versus the reliability of the solution. The current technological level of modern computers allows differential equations like the RKEs to be solved "quickly" with any of the mentioned methods. That is, a single iteration requires little computational time. However, real world problems require many iterations of solving the same system of differential equation. Reactors in reality have spatial dependence, are not homogeneous, and factors like temperature and material flow for moderator/coolant will greatly influence the conditions of the system of equations. The equation must be solved over many time steps, over a spatial grid. The size of the grid must capture the geometry of the reactor, and gradients of neutron density throughout the reactor.

Finding a method that solves the PRKEs quickly and accurately can potentially reduce the amount of time it takes to solve reactor kinetics diffusion and transport problems. This is the main motivation for the search and analysis of new methods of an otherwise straightforward problem.

I.C. Problem Statement

Exponential moment methods can be implemented in an algorithm that will solve an approximation of the PRKEs. This research aims to characterize the method and evaluate the fidelity of the solution as a function of step size. This method can be expanded using an error control scheme to further enhance the value of computational time.

I.D. Objectives

I.D.1 Validation of Solution

The method should be able to produce the correct solution to a certain degree specified by the tolerances given as an input. The solution can be compared to the solution produced by a trusted mathematics package, such as Mathematica. Certain initial conditions will produce trivial situations that the code should accurately replicate. These include a critical system at steady state and a subcritical system with a source. Standard problems, such as linear reactivity

should also be explored. Problems that have special conditions that require a more rigorous approach warrant their own objective.

I.D.2 Controlled Error Accumulation

The nature of coded solutions results in several sources of error. The error created by the algorithm choices should be observable as it propagates through multiple time steps. This allows the prediction and quantification of error of the method itself.

I.D.3 Solution for Sinusoidal Reactivity

The algorithm should be able to handle any realistic and well-behaved form of reactivity. Sinusoidal reactivity is an example that exists in reality. Sinusoidal reactivity can occur in actual reactors through periodic rod movement. Mathematically, this creates a complication in our approach to solving the PRKEs that should be addressed. The solution should be evaluated for accuracy and performance. Any additional sources of error should be quantified and documented.

I.D.4 Convergence Performance Evaluation

Our method should be comparable to other common methods of solving the PRKEs for accuracy and feasibility. One attribute of methods that step through the solution share in common is the dependence of error upon step size. The order of convergence can be determined through the analysis of the error and compared to other methods if desired. This order could change for variations of the problem, such as the sinusoidal reactivity condition.

I.D.5 Approximate Solution using Large Time Steps

The performance of the method should be evaluated for large time steps. For large time steps, error is inevitable but quantifiable. If the error is tolerable, one can quickly observe the general behavior of a solution for many reasons, which can be useful for back-of-the-envelop calculation, or more under certain circumstances. In essence, shows the ability to mitigate the effects of stiffness. If the mechanism that defines the convergence is mapped out to large enough step sizes, one can determine if large step sizes are feasible in situations where accuracy is not as valued as computational cost.

I.D.6 Error Control Algorithm with Adaptive Time Steps

Error control allows computation time to be saved during portions of the problem where extra time steps are wasted due to the simplicity of the problem

conditions in those regions. A functional error control scheme should have the ability to modify the error generated by adjusting tolerances. The step sizes should be appropriate to the behavior of the problem at that time. An especially robust error control scheme can find the solution through problematic regions of the given conditions that are especially stiff. The test case of interest is a reactor that has a reactivity that increases over time and passes through prompt criticality.

I.E. Summary of Approach

The system of differential equations given in (2) and (3) can be converted into integral equations. The neutron density $n(t)$ is approximated by an exponential form over a time step of

$$n(t) = ae^{\alpha t}. \tag{5}$$

By substituting this assumption and taking the 0^{th} and 1^{st} temporal moment, the integral equations can be written with each section matching the form of an exponential moment function. An exponential moment function is the solution to a specific form of integral, which can be evaluated recursively. The general definition is

$$M_n(x_{1:k}) \equiv \int_0^1 du_1(1-u_1)^n e^{-x_1 u_1} \int_0^{u_1} du_2 e^{(x_1-x_2)u_2} ... \int_0^{u_{k-1}} du_k e^{(x_{k-1}-x_k)u_k} . \tag{6}$$

The 0^{th} and 1^{st} temporal moments of equation (5) can be set equal to the 0^{th} and 1^{st} temporal moment of the neutron density RKE and there are now two equations with two unknowns, a and α. Solving for these two variables and substituting back into our original system of equations, one can find the approximate neutron and precursor density over the time step. Adjusting the time step size will determine the amount of error in the solution as error has a strong dependence on step size for numerical methods such as this one.

II. REVIEW OF REACTOR KINETIC EQUATIONS

II.A. Introduction to Reactor Equations

Reactor dynamics is the study of the phenomena in a nuclear reactor system with knowledge of various aspects of the reactor, such as temperature, control rod status, and neutron poison build up. These listed aspects affect the reactivity and if the functional form of the reactivity is determined, the problem can be reduced to a reactor kinetics problem. The RKEs have the following independent variables: space, energy, direction of motion of neutrons, and time.

Reactor equations are balance equations. That is, they conserve neutrons within the problem by considering each mechanism in which neutrons are created and lost. Neutrons are created through fission or added through a source, and both of these represent terms in a reactor equation. Other terms represent neutrons that are lost through absorption, or exiting the grid entirely.

II.B. Transport and Diffusion Equations

The *neutron transport kinetics equation* is the neutron balance that takes into account the basic reactor kinetic independent variables. The equation explicitly creates terms to represent the phenomena within a reactor system and is commonly used to determine the behavior of reactor cores or neutrons beams. The equation is

$$\frac{1}{v(E)}\frac{\partial\psi(r,E,\hat{\Omega},t)}{\partial t}+\hat{\Omega}\cdot\nabla\psi(r,E,\hat{\Omega},t)+\Sigma_t(r,E,t)\,\psi(r,E,\hat{\Omega},t)=$$
$$\frac{\chi_f(E)}{4\pi}\int_0^\infty dE'\,\nu_f(E')\Sigma_f(r,E',t)\phi(r,E',t)+\sum_{i=1}^{N}\frac{\chi_{d_i}(E)}{4\pi}\lambda_i C_i(r,t)+ \tag{7}$$
$$\int_{4\pi}d\Omega'\int_0^\infty dE'\,\Sigma_s(r,E'\rightarrow E,\hat{\Omega}'\rightarrow\hat{\Omega},t)\,\psi(r,E',\hat{\Omega}',t)+S(r,E,\hat{\Omega},t)$$

where the independent variables are

r	Position vector (x,y,z)
E	Energy
$\hat{\Omega}$	Unit Vector in direction of motion
t	Time

and the various terms within the equation are:

$v(E)$	Neutron speed
$\psi(r,E,\hat{\Omega},t)\,dr\,dE\,d\Omega$	Angular neutron flux
$\phi(r,E,t)\,dr\,dE$	Scalar neutron flux
ν_f	Average number of neutrons produced per fission
χ_f	Density function of neutrons exiting with energy E (fission)
χ_{d_i}	Density function of neutrons exiting with energy E (delayed)

$\Sigma_t(r,E,t)$	Macroscopic total cross section
$\Sigma_f(r,E',t)$	Macroscopic fission cross section
$\Sigma_s(r,E' \rightarrow E,\hat{\Omega}' \rightarrow \hat{\Omega},t)$	Macroscopic scattering cross section
N	Number of precursor groups
λ_i	Decay constant for group i
$C_i(r,t)$	Precursor density for group i
$S(r,E,\hat{\Omega},t)$	Source term

Often the angular dependence isn't known and the basic transport equation can prove to be difficult to solve. Through some approximation and integration with respect to $\hat{\Omega}$, one can transform the transport equation into the diffusion equation, which is often sufficient for solving realistic problems.

$$\begin{aligned} &-\nabla \cdot D(r,E,t)\nabla\phi(r,E,t) + \Sigma_r(r,E,t)\phi(r,E,t) = \\ &\int_0^\infty \partial E' \Sigma_s(r,E' \rightarrow E,t)\phi(r,E',t) + \sum_{i=1}^{N} \frac{\chi_{d_i}(E)}{4\pi}\lambda_i C_i(r,t) \\ &\chi_f(E)\int_0^\infty \partial E' \nu_f(E')\Sigma_f(r,E',t)\phi(r,E',t) + S(r,E,t) \end{aligned} \tag{8}$$

Note that the scalar flux is used throughout the equation and the new variable D is the diffusion coefficient which is related to the cross sections. The diffusion equation is much more managable than the full transport equation, but still retains a certain degree of complexity [2].

II.C. Point Reactor Kinetic Equations

Additional simplifications to the diffusion equation will yield the PRKEs, which can be used to test novel numerical techniques for studying reactor behavior before applying those techniques to the more complicated diffusion or transport equations. The first simplification is integrating energy into groups, or even a single energy group that represents the average behavior across all energies. This will remove energy dependence. Another simplification is assuming an infinite, homogeneous region of interest where the spatial dependence of the problem is removed. Knowing that each neutron that leaves a region will be replaced symmetrically by another neutron, one can assume that any given point in this reactor will mimic all other points. This is known as the point reactor. The diffusion equation is simplified into this form by merging the gain and loss terms as the reactivity of the reactor, and splitting the delayed neutron precursors into their own additional equations. These equations were shown in (2) and (3). Note that the flux has now been replaced by neutron density, all loss and gain mechanisms have been contained within the reactivity and the equation has been split according to precursor groups.

Solving the PRKEs with respect to time has its own set of difficulties. The general form of reactivity with time, and the number of delayed precursor groups does not allow a closed form solution. Instead, iterative numerical methods are

used. However, each delayed group presents decay constants that are different by orders of magnitude (Table 3). The result is a very stiff system of differential equations. Various standard methods for solving such differential equations can be shown to be ineffective for this specific set. Forward Euler is an explicit method that is very easy to implement. The nature of forward Euler and most other explicit methods requires very small step sizes in order to deal with stiff problems. The result is the requirement of many steps without a guarantee that the end solution is reliable. To compare, backwards Euler is an implicit methods that is not prone to numerical instability. For non-linear problems, such as the PRKEs, backwards Euler can be very expensive and require a suitable root finding algorithm. These methods are not normally considered for use. The solution will still develop error linearly according to step size. There are robust explicit and implicit methods that exchange computational cost for a higher order of convergence, such as various Runge-Kutta methods, but even these are hindered by the stiffness of the problem.

Overall, any of these various techniques are generally considered acceptable for solving a single PRKEs problem because modern computing allows us to solve many small time steps or compute complex algorithms in a tolerable time frame. However, as shown in the review of RKEs, the PRKEs are a greatly simplified adaptation of complex equations that better fit reality. Once other variables are considered, it is no longer acceptable or affordable to solve the problem with

these standard methods. A spatial parameter alone would multiply the amount of calculation time by the grid size. Any time saved in calculation would also be multiplied. Creating a robust solution to PRKE can provide insight to managing the computational requirements of other RKEs. The robust solution of interest is using exponential moment methods in order to solve an approximation of the PRKE. Exponential moment methods specify the solution of specific integrals with a solution that can be calculated recursively. Differential equations can be converted to integral equations and then converted into moment functions where the complexity of the algorithm and step size requirement can be compared with other methods. These results can be used to motivate exploration of exponential moment methods in transport and diffusion equations.

III. OVERVIEW OF EXPONENTIAL MOMENT FUNCTIONS

III.A. Definition and Properties of Exponential Moment Functions

This review section is a presentation of key information from the book *Exponential Moment Methods*, currently in draft [1]. The general form of an exponential moment function is

$$\mathbf{M}_n\left(x_{1:k}\right) \equiv \int_0^1 du_1 \left(1-u_1\right)^n e^{-x_1 u_1} \int_0^{u_1} du_2 e^{\left(x_1-x_2\right)u_2} \ldots \int_0^{u_{k-1}} du_k e^{\left(x_{k-1}-x_k\right)u_k} . \tag{9}$$

Two defining attributes of any exponential moment function are its order and rank. The general equation above is of order n, and rank k. The order of the function is the power to which $\left(1-u_1\right)$ is raised. The rank is the number of arguments presented in the function. Exponential moment functions are orderless; that is, the value of the function is invariant under permutations of its arguments. Exponential moment functions are also always positive and monotonically decreasing. Mathematical manipulation and perhaps a few approximations can be used to turn many equations into the form of the right hand side of equation (9). Exponential moment functions use recursion to create an algebraic equivelant of the integral. In general, the needed recursion will reduce a moment function into many moment functions that are order 0 and rank 1.

The main mechanism for reduction in rank is

$$\mathcal{M}_n\left(x_{1:k}\right) = \frac{\mathcal{M}_n\left(x_{1:k-2}, x_{k-1}\right) - \mathcal{M}_n\left(x_{1:k-2}, x_k\right)}{x_k - x_{k-1}}. \tag{10}$$

Using the orderless feature of exponential moment functions, the arguments can be sorted in increasing order. Then by applying the recursive formula using the first and last argument, one can force the denominator to be as large as possible. Thus the form of equation (10) used is

$$\mathcal{M}_n\left(x_{1:k}\right) = \frac{\mathcal{M}_n\left(x_{2:k}\right) - \mathcal{M}_n\left(x_{1:k-1}\right)}{x_1 - x_k} \tag{11}$$

where x is sorted in decreasing order.

At a certain threshold, special treatment can be used for arguments that become too close together indicating possible issues. Moment functions are poorly conditioned for arguments that differ by small amounts. In such a case, the division leads to catastrophic cancellation and a major loss of precision. A certain approach to the problem can pinpoint this problem when it occurs and special treatment can be used to prevent such issues.

Once all exponential moment functions are of rank 1, the order must be reduced. Backwards recurrence on order is done using

$$\mathcal{M}_n\left(x\right) = \frac{1 - x\mathcal{M}_{n+1}\left(x\right)}{n+1} \qquad \forall n \geq 0\,. \tag{12}$$

After enough applications of the recurrence relation, all exponential moment functions are reduced to rank 1 and order 0. The basic algebraic definition of such an exponential moment function is

$$\mathcal{M}_0\left(x\right) = \frac{1 - e^{-x}}{x}. \tag{13}$$

An exponential moment function can always be reduced to an algebraic formula that is just subtractions and divisions. Depending on the arguments, some computational issues may arrive and are addressed accordingly. Additionally, some series expansions and other properties of exponential moment functions can be used to accelerate the computation or decrease the loss of precision and are implemented. These methods are explained in *Exponential Moment Methods* [1].

IV. APPLICATION OF EXPONENTIAL MOMENT METHODS TO THE SOLUTION OF POINT REACTOR KINETIC EQUATIONS

This research seeks a numerical scheme to solve the PRKEs. In *Exponential Moment Methods*, Mathews explains an application of exponential moment functions to point-reactor kinetics [1]. This method is replicated throughout this section.

Exponential moment methods are applied to integral equations and thus converting the PRKEs into integral equations is our initial step. Starting with the standard equations (2) and (3), integrating factors are used. For the i^{th} second PRKE:

$$\begin{aligned}
&\frac{dc_i\left(t\right)}{dt}e^{\lambda_i t} + \lambda_i c_i\left(t\right)e^{\lambda_i t} = \frac{\beta_i}{\Lambda}n\left(t\right)e^{\lambda_i t} \\
&\frac{d}{dt}\left(c_i\left(t\right)e^{\lambda_i t}\right) = \frac{\beta_i}{\Lambda}n\left(t\right)e^{\lambda_i t} \\
&\int_0^{t'} d\left(c_i\left(t\right)e^{\lambda_i t}\right) = \int_0^{t'} dt''\frac{\beta_i}{\Lambda}n\left(t''\right)e^{\lambda_i t''} \\
&c_i\left(t'\right)e^{\lambda_i t'} - c_i\left(0\right) = \frac{\beta_i}{\Lambda}\int_0^{t'} e^{\lambda_i t''}n\left(t''\right)dt'' \\
&c_i\left(t'\right) = c_i\left(0\right)e^{-\lambda_i t'} + \frac{\beta_i}{\Lambda}\int_0^{t'} e^{-\lambda_i\left(t'-t''\right)}n\left(t''\right)dt''
\end{aligned} \qquad (14)$$

The first equation cannot be treated in a similar fashion due to the fact that the reactivity is not constant. A constant is substituted for the sake of forming an integral equation and the deviation from the chosen constant will have to be

addressed. The average reactivity, $\overline{\rho}$, over the time step of interest is used as the constant. The deviations from the average are

$$\delta\rho\left(t\right) = \rho\left(t\right) - \overline{\rho} \tag{15}$$

and the average reactivity as

$$\overline{\rho} = \int_0^{\Delta t} \rho\left(t\right)\frac{dt}{\Delta t}. \tag{16}$$

However, reactivity itself isn't a constant and cannot be used for an integrating factor in the first reactor kinetics equation. A new constant and deviation factor is defined.

$$\begin{aligned} \overline{\kappa} &= \frac{\beta - \overline{\rho}}{\Lambda} \\ \delta\kappa\left(t\right) &= \frac{\delta\rho(t)}{\Lambda} \end{aligned} \tag{17}$$

Using these the first PRKE can be turned into an integral equation by

$$\begin{aligned} \frac{dn(t)}{dt} &= \left(\frac{\rho(t) - \beta}{\Lambda}\right)n(t) + \sum_i \lambda_i c_i(t) + S(t) \\ \frac{dn(t)}{dt} &= \left(\frac{\delta\rho(t) + \overline{\rho} - \beta}{\Lambda}\right)n(t) + \sum_i \lambda_i c_i(t) + S(t) \\ \frac{dn(t)}{dt} &= \left(\frac{\overline{\rho} - \beta}{\Lambda}\right)n(t) + \sum_i \lambda_i c_i(t) + S(t) + \frac{\delta\rho(t)}{\Lambda}n(t) \\ \frac{dn(t)}{dt} + \overline{\kappa}n(t) &= \sum_i \lambda_i c_i(t) + S(t) + \delta\kappa\left(t\right)n(t) \end{aligned} \tag{18}$$

,

then solve using the integrating factor $e^{\overline{\kappa}t}$.

$$\frac{dn(t)}{dt}e^{\overline{\kappa}t} + \overline{\kappa}n(t)e^{\overline{\kappa}t} = \sum_i \lambda_i c_i(t)e^{\overline{\kappa}t} + S(t)e^{\overline{\kappa}t} + \delta\kappa\left(t\right)n(t)e^{\overline{\kappa}t}$$

$$\frac{d}{dt}\left(n(t)e^{\overline{\kappa}t}\right) = \sum_i \lambda_i c_i(t)e^{\overline{\kappa}t} + S(t)e^{\overline{\kappa}t} + \delta\kappa\left(t\right)n(t)e^{\overline{\kappa}t}$$

$$\int_0^t d\left(n(t)e^{\overline{\kappa}t}\right) = \int_0^t dt'\left(\sum_i \lambda_i c_i(t')e^{\overline{\kappa}t'} + S(t')e^{\overline{\kappa}t'} + \delta\kappa\left(t'\right)n(t')e^{\overline{\kappa}t'}\right) \tag{19}$$

$$n(t) = n(0)e^{-\overline{\kappa}t} + \int_0^t dt' \sum_i \lambda_i c_i(t')e^{-\overline{\kappa}\left(t-t'\right)} + \int_0^t e^{-\overline{\kappa}\left(t-t'\right)}S\left(t'\right)dt'$$
$$+\int_0^t e^{-\overline{\kappa}\left(t-t'\right)}\delta\kappa\left(t'\right)n\left(t'\right)dt'$$

Finally, the dependence on the precursor densities is eliminated by substituting the second RPKE in the first.

$$n\left(t\right) = n\left(0\right)e^{-\overline{\kappa}t} + \sum_i \lambda_i c_i\left(0\right)\int_0^t e^{-\overline{\kappa}\left(t-t'\right)}e^{-\lambda_i t'}dt'$$
$$+\int_0^t e^{-\overline{\kappa}\left(t-t'\right)}S\left(t'\right)dt' + \int_0^t e^{-\overline{\kappa}\left(t-t'\right)}\delta\kappa\left(t'\right)n\left(t'\right)dt' \tag{20}$$
$$+\sum_i \frac{\lambda_i\beta_i}{\Lambda}\int_0^t e^{-\overline{\kappa}\left(t-t'\right)}\left[\int_0^{t'} e^{-\lambda_i\left(t'-t''\right)}n\left(t''\right)dt''\right]dt' \ .$$

At this point, only one equation remains with everything known, except for the neutron density which is present within itself.

IV.A. Neutron Density Approximation

An exponential form is assumed for the neutron density within a time step.

$$\tilde{n}\left(t\right) = ae^{\alpha t} \quad \text{for} \quad 0 \le t \le \Delta t \tag{21}$$

The exponential form is chosen for its nonlinearity and implicit nature. Linear, implicit, single-step methods cannot be A-stable (approach 0 to solutions of differential equations that approach 0 as t approaches ∞) and more than

second-order accurate. The exponential form has two degrees of freedom, a and α that are to be determined iteratively. Equation (21) is substituted into (14) and (20) to obtain:

$$c_i\left(t'\right) = c_i\left(0\right)e^{-\lambda_i t'} + a\frac{\beta_i}{\Lambda}e^{-\lambda_i t'}\int_0^{t'} e^{\left(\lambda_i+\alpha\right)t''}dt'' \tag{22}$$

and

$$\begin{aligned} n\left(t\right) = n\left(0\right)e^{-\overline{\kappa}t} + \sum_i \lambda_i c_i\left(0\right)\int_0^t e^{-\overline{\kappa}\left(t-t'\right)}e^{-\lambda_i t'}dt' \\ +\int_0^t e^{-\overline{\kappa}\left(t-t'\right)}S\left(t'\right)dt' + ae^{-\overline{\kappa}t}\int_0^t e^{\left(\overline{\kappa}+\alpha\right)t'}\delta\kappa\left(t'\right)dt' \\ +a\sum_i \frac{\lambda_i\beta_i}{\Lambda}\int_0^t e^{-\overline{\kappa}\left(t-t'\right)}\left[\int_0^{t'} e^{-\lambda_i\left(t'-t''\right)}e^{\alpha t''}dt''\right]dt' \ . \end{aligned} \tag{23}$$

Currently equation (23) would be the first Picard iteration if a and α were specified. One may note that equation (23) can be substituted into (14) and (20) to obtain what may be a higher order solution. The form of such a solution would be expensive to evaluate. However, occasionally doing so would allow us to find some error margin in order to compare against a threshold for error control in an adaptive step size method. The use of Picard iteration is discussed in VIII.A. For now, a and α must be determined in a different manner. Moment matching is a robust method of specifying unknowns within an expression. Moments allow us to choose weighting factors for both convenience of solution and to capture the behavior of our expressions. To determine a and α, the 0^{th} and 1^{st} temporal moments of equations (21) and (23) are set equal to each other. The weight function is chosen to allow the integrals to be

manipulated into exponential moment functions. This tactic is the basic approach used for all integrals and reduces the portions of iteration that would have to be done otherwise. For the 0^{th} moment:

$$\begin{aligned}
\tilde{n}_0 &\equiv \int_0^{\Delta t} \tilde{n}(t)\frac{dt}{\Delta t} \\
\tilde{n}_0 &= a\int_0^{\Delta t} e^{\alpha t}\,\frac{dt}{\Delta t} \\
\tilde{n}_0 &= a\int_0^{1} e^{-(-\alpha\Delta t)u}du \\
\tilde{n}_0 &= a\mathcal{M}_0\Big(-\alpha\Delta t\Big)
\end{aligned} \tag{24}$$

and for the 1^{st} moment:

$$\begin{aligned}
\tilde{n}_1 &\equiv \int_0^{\Delta t}\left(1-\frac{t}{\Delta t}\right)\tilde{n}(t)\frac{dt}{\Delta t} \\
\tilde{n}_1 &= a\int_0^{1}\Big(1-u\Big)e^{-(-\alpha\Delta t)u}du \\
\tilde{n}_1 &= a\mathcal{M}_1\Big(-\alpha\Delta t\Big).
\end{aligned} \tag{25}$$

One may note that if one assumed a different form for the neutron density, one could take more moments and choose the weighting factor accordingly in order to fit the new form. These moments must be equated to the moments of (23) which after some manipulation are

$$\begin{aligned}
n_0\Big(a,\alpha\Delta t\Big) = n\Big(0\Big)\mathcal{M}_0\Big(\overline{\kappa}\Delta t\Big) &+ \sum_i c_i\Big(0\Big)\Big(\lambda_i\Delta t\Big)\mathcal{M}_0\Big(\overline{\kappa}\Delta t,\lambda_i\Delta t\Big) \\
&+a\sum_i \beta_i\Big(\lambda_i\Delta t\Big)\left(\frac{\Delta t}{\Lambda}\right)\mathcal{M}_0\Big(\overline{\kappa}\Delta t,\lambda_i\Delta t,-\alpha\Delta t\Big) \\
&+\int_0^{\Delta t} e^{-\overline{\kappa}t}\left[\int_0^t e^{\left(\overline{\kappa}-0\right)t'}S\Big(t'\Big)dt'\right]\frac{dt}{\Delta t} \\
&+a\int_0^{\Delta t} e^{-\overline{\kappa}t}\left[\int_0^t e^{\left(\overline{\kappa}+\alpha\right)t'}\delta\kappa\Big(t'\Big)dt'\right]\frac{dt}{\Delta t}.
\end{aligned} \tag{26}$$

and

$$
\begin{aligned}
n_1\left(a,\alpha\Delta t\right) = n\left(0\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t\right) &+ \sum_i c_i\left(0\right)\left(\lambda_i\Delta t\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t,\lambda_i\Delta t\right) \\
&+ a\sum_i \beta_i\left(\lambda_i\Delta t\right)\left(\frac{\Delta t}{\Lambda}\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t,\lambda_i\Delta t,-\alpha\Delta t\right) \\
&+ \int_0^{\Delta t}\left(1-\frac{t}{\Delta t}\right)e^{-\overline{\kappa}t}\left[\int_0^t e^{\left(\overline{\kappa}-0\right)t'}S\left(t'\right)dt'\right]\frac{dt}{\Delta t} \\
&+ a\int_0^{\Delta t}\left(1-\frac{t}{\Delta t}\right)e^{-\overline{\kappa}t}\left[\int_0^t e^{\left(\overline{\kappa}+\alpha\right)t'}\delta\kappa\left(t'\right)dt'\right]\frac{dt}{\Delta t}.
\end{aligned}
\tag{27}
$$

The form of the source and the reactivity will vary from problem to problem. The method is designed to allow the conversion of these factors into exponential moment functions, but also to encompass various realistic forms that the source and reactivity may undertake.

IV.B. Source Term

In general, the source of a reactor system is usually constant. To add some generality and flexibility, the solution is designed to function for any polynomial source. The following general form is used:

$$
S\left(t\right) = \sum_{p=0}^{P_{source}} s_p\left(\frac{t}{\Delta t}\right)^p . \tag{28}
$$

This form is then substituted into the source term of equation (26) and (27). The source integral is transformed into an exponential moment function, but the means in which that is done is not readily apparent. A short derivation should

shed some insight on the matter. For the 0^{th} moment, the polynomial source term is substituted into the 0^{th} moment source integral.

$$n_0^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^{\Delta t} e^{-\overline{\kappa} t} \left[\int_0^t e^{\left(\overline{\kappa}-0\right)t'} \left(\frac{t'}{\Delta t} \right)^p \frac{dt'}{\Delta t} \right] \frac{dt}{\Delta t} \tag{29}$$

Applying the following change of variables:

$$\begin{aligned} u_1 &= \frac{t}{\Delta t} \\ u_2 &= \frac{t'}{\Delta t} \end{aligned} \tag{30}$$

$$n_0^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 du_1 e^{-\left(\overline{\kappa}\Delta t\right)u_1} \int_0^{u_1} du_2 \, e^{\left(\overline{\kappa}\Delta t-0\right)u_2} {u_2}^p \tag{31}$$

.

Reverse the sequence of integration noting that the bounds must change in order to integrate correctly.

$$n_0^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 du_2 {u_2}^p e^{\left(\overline{\kappa}\Delta t-0\right)u_2} \int_{u_2}^1 du_1 \, e^{-\left(\overline{\kappa}\Delta t\right)u_1} \tag{32}$$

Apply another change of variables in order to get the bounds to match the form of an exponential moment function.

$$\begin{aligned} 1 - v_2 &= u_2 \\ 1 - v_1 &= u_1 \end{aligned} \tag{33}$$

$$n_0^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 dv_2 \left(1 - v_2\right)^p e^{\left(\overline{\kappa}\Delta t-0\right)\left(1-v_2\right)} \int_0^{v_2} dv_1 \, e^{-\left(\overline{\kappa}\Delta t\right)\left(1-v_1\right)} \tag{34}$$

Finally, simplify the exponents and gather the arguments in the form of an exponential moment function.

$$
\begin{aligned}
n_0^{\text{poly.source}} &= \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 dv_2 \left(1-v_2\right)^p e^{-\left(\overline{\kappa}\Delta t\right)v_2} \int_0^{v_2} dv_1 \, e^{\left(\overline{\kappa}\Delta t-0\right)v_1} \\
n_0^{\text{poly.source}} &= \sum_{p=0}^{P_{source}} s_p \Delta t \, \mathcal{M}_p\left(\overline{\kappa}\Delta t, 0\right)
\end{aligned}
\tag{35}
$$

When creating these algorithms to minimize calculation cost, mathematical manipulations are applied to reduce the number of total calculations required. Recursive techniques benefit greatly from reduction of iterations required and a certain identity of exponential moment functions allows us to immediately save one iteration. This identity is

$$
\mathcal{M}_n\left(x_{1:j}, 0\right) = \frac{\mathcal{M}_{n+1}\left(x_{1:j}\right)}{n+1} \, . \tag{36}
$$

Applying (36) to our derivation gives us our solution to the 0^{th} moment for polynomial sources.

$$
n_0^{\text{poly.source}} = \sum_{p=0}^{P_{source}} \frac{s_p \Delta t}{p+1} \mathcal{M}_{p+1}\left(\overline{\kappa}\Delta t\right) \tag{37}
$$

The 1^{st} moment requires a little more effort. As before, the polynomial source is substituted into the 1^{st} moment source integral.

$$
n_1^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^{\Delta t} \left(1-\frac{t}{\Delta t}\right) e^{-\overline{\kappa}t} \left[\int_0^t e^{\left(\overline{\kappa}-0\right)t'} \left(\frac{t'}{\Delta t}\right)^p \frac{dt'}{\Delta t}\right] \frac{dt}{\Delta t} \tag{38}
$$

Applying the same change of variables shown in (30).

$$
n_1^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 du_1 \left(1-u_1\right) e^{-\left(\overline{\kappa}\Delta t\right)u_1} \int_0^{u_1} du_2 \, e^{\left(\overline{\kappa}\Delta t-0\right)u_2} {u_2}^p \tag{39}
$$

Reverse the sequence of integration.

$$n_1^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 du_2 {u_2}^p e^{\left(\overline{\kappa}\Delta t - 0\right)u_2} \int_{u_2}^1 du_1 \left(1 - u_1\right) e^{-\left(\overline{\kappa}\Delta t\right)u_1} \tag{40}$$

Apply the same change of variables shown in (33).

$$n_1^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 dv_2 \left(1 - v_2\right)^p e^{\left(\overline{\kappa}\Delta t - 0\right)\left(1 - v_2\right)} \int_0^{v_2} dv_1 \, e^{-\left(\overline{\kappa}\Delta t\right)\left(1 - v_1\right)} v_1 \tag{41}$$

The v_1 term is problematic and doesn't allow one to produce the form of an exponential moment function easily. Using some creativity, this variable can be changed into an additional argument of a rank 3 exponential moment function

$$\begin{aligned} n_1^{\text{poly.source}} &= \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 dv_2 \left(1 - v_2\right)^p e^{\left(\overline{\kappa}\Delta t - 0\right)\left(1 - v_2\right)} \int_0^{v_2} dv_1 \, e^{-\left(\overline{\kappa}\Delta t\right)\left(1 - v_1\right)} \int_0^{v_1} dv_0 \\ n_1^{\text{poly.source}} &= \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 dv_2 \left(1 - v_2\right)^p e^{-\left(\overline{\kappa}\Delta t\right)v_2} \int_0^{v_2} dv_1 \, e^{\left(\overline{\kappa}\Delta t - 0\right)v_1} \int_0^{v_1} dv_0 \, e^{\left(0 - 0\right)v_0} \end{aligned} \tag{42}$$

Thus, the form of an exponential moment function reveals itself.

$$n_1^{\text{poly.source}} = \sum_{p=0}^{P_{source}} s_p \Delta t \, \mathcal{M}_p\left(\overline{\kappa}\Delta t, 0, 0\right) \tag{43}$$

Using identity (36) twice, the solution elegantly simplifies itself to

$$n_1^{\text{poly.source}} = \sum_{p=0}^{P_{source}} \frac{s_p \Delta t}{\left(p+1\right)\left(p+2\right)} \mathcal{M}_{p+2}\left(\overline{\kappa}\Delta t\right). \tag{44}$$

After a and α are determined through moment matching, the neutron density will be calculated. One of the terms within the neutron density, as

shown in (23), is the source term. The source term here is an integral which can be evaluated using exponential moment methods. The isolated integral is

$$n^{\text{source}}\left(\Delta t\right) = \int_0^{\Delta t} e^{-\overline{\kappa}\left(\Delta t - t\right)} S\left(t\right) dt \tag{45}$$

Substituting in the polynomial source assumption and multiplying and dividing by Δt gives us

$$n^{\text{poly. source}}\left(\Delta t\right) = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^{\Delta t} e^{-\left(\overline{\kappa}\Delta t\right)\left(1 - t/\Delta t\right)} \left(\frac{t}{\Delta t}\right)^p \frac{dt}{\Delta t}. \tag{46}$$

Apply the following change of variables:

$$v = \frac{t}{\Delta t} \tag{47}$$

$$n^{\text{poly. source}}\left(\Delta t\right) = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 e^{-\left(\overline{\kappa}\Delta t\right)\left(1 - v\right)} \left(v\right)^p dv \tag{48}$$

and another change of variables:

$$1 - u = v \tag{49}$$

$$n^{\text{poly. source}}\left(\Delta t\right) = \sum_{p=0}^{P_{source}} s_p \Delta t \int_0^1 \left(1 - u\right)^p e^{-\left(\overline{\kappa}\Delta t\right)u} du \tag{50}$$

,

and once again the form of an exponential moment function is revealed:

$$n^{\text{poly. source}}\left(\Delta t\right) = \sum_{p=0}^{P_{source}} s_p \Delta t \, \mathcal{M}_p\left(\overline{\kappa}\Delta t\right)$$

.

IV.C. Reactivity Term

The same treatment must be done to the reactivity term, $\delta\kappa$. Unlike the source term, reactivity can take many forms. As always, it is our choice to pick a method that is both robust and convenient. The nature of δk compared to the actual reactivity, ρ, must also be taken into account. The function δk is either zero throughout a time step for constant reactivity, or it crosses zero at least once. Our approximation should have this capability as well. A polynomial approximation can be appropriate assuming the time-step is chosen to allow for such a fit. Then any form of reactivity can be transformed into a polynomial approximation through moment matching. Let us assume the same polynomial form for the δk term of order P_ρ.

$$\delta\kappa\left(t'\right) = \sum_{p=0}^{P_\rho} \delta\kappa_p \left(\frac{t'}{\Delta t}\right)^p \tag{51}$$

This form is then substituted into the integral containing $\delta\kappa$ in equations (26) and (27). The $\delta\kappa$ term is changed into an exponential moment function, but the means in which that is done, like the source term, is not readily apparent.

As with the source term, the polynomial δk term is substituted into the 0^{th} moment reactivity integral.

$$n_0^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \int_0^{\Delta t} e^{-\overline{\kappa} t} \left[\int_0^t e^{\left(\overline{\kappa}+\alpha\right)t'} \left(\frac{t'}{\Delta t} \right)^p \frac{dt'}{\Delta t} \right] \frac{dt}{\Delta t} \tag{52}$$

Apply the change of variables given in (30).

$$n_0^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \int_0^1 du_1 \, e^{-\left(\overline{\kappa}\Delta t\right)u_1} \int_0^{u_1} du_2 \, e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)u_2} {u_2}^p \tag{53}$$

Reverse the sequence of integration.

$$n_0^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \int_0^1 du_2 {u_2}^p e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)u_2} \int_{u_2}^1 du_1 \, e^{-\left(\overline{\kappa}\Delta t\right)u_1} \tag{54}$$

Apply the change of variables given in (33).

$$n_0^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \int_0^1 dv_2 \left(1-v_2\right)^p e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)\left(1-v_2\right)} \int_0^{v_2} dv_1 \, e^{-\left(\overline{\kappa}\Delta t\right)\left(1-v_1\right)} \tag{55}$$

Simplify exponents and gather arguments.

$$n_0^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t e^{\alpha\Delta t} \int_0^1 dv_2 \left(1-v_2\right)^p e^{-\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)v_2} \int_0^{v_2} dv_1 \, e^{\left(\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)-\alpha\Delta t\right)v_1} \tag{56}$$

Now the form of an exponential moment function reveals itself as

$$n_0^{\delta\kappa}\left(a,\alpha\Delta t\right) = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \, e^{\alpha\Delta t} \mathcal{M}_p\left(\overline{\kappa}\Delta t+\alpha\Delta t,\alpha\Delta t\right). \tag{57}$$

Similarly, the same method is applied to the 1st moment. The polynomial δk term is substituted into the 1st moment reactivity integral.

$$n_1^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \int_0^{\Delta t} \left(1-\frac{t}{\Delta t}\right) e^{-\overline{\kappa} t} \left[\int_0^t e^{\left(\overline{\kappa}+\alpha\right)t'} \left(\frac{t'}{\Delta t} \right)^p \frac{dt'}{\Delta t} \right] \frac{dt}{\Delta t} \tag{58}$$

Apply the change of variables given in (30).

$$n_1^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \int_0^1 du_1 \left(1-u_1\right) e^{-\left(\overline{\kappa}\Delta t\right)u_1} \int_0^{u_1} du_2\, e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)u_2} {u_2}^p \tag{59}$$

Reverse the sequence of integration.

$$n_1^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \int_0^1 du_2 {u_2}^p e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)u_2} \int_{u_2}^1 du_1 \left(1-u_1\right) e^{-\left(\overline{\kappa}\Delta t\right)u_1} \tag{60}$$

Apply the change of variables given in (33).

$$n_1^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t \int_0^1 dv_2 \left(1-v_2\right)^p e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)\left(1-v_2\right)} \int_0^{v_2} dv_1\, e^{-\left(\overline{\kappa}\Delta t\right)\left(1-v_1\right)} v_1 \tag{61}$$

As with the source 1st moment derivation, there is an extra v_1 term. The same trick can be used again to convert it into an extra argument.

$$\begin{aligned} n_1^{\text{poly.}\delta\kappa} = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)} e^{-\left(\overline{\kappa}\Delta t\right)} \Bigg[& \int_0^1 dv_2 \left(1-v_2\right)^p e^{-\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)v_2} \\ & \times \int_0^{v_2} dv_1\, e^{\left(\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)-\alpha\Delta t\right)v_1} \int_0^{v_1} dv_0\, e^{\left(\alpha\Delta t-\alpha\Delta t\right)v_0} \Bigg]. \end{aligned} \tag{62}$$

Now the form of an exponential moment function reveals itself as

$$n_1^{\delta\kappa}\left(a,\alpha\Delta t\right) = \sum_{p=0}^{P_\rho} \delta\kappa_p \Delta t\, e^{\alpha\Delta t} \mathcal{M}_p\left(\overline{\kappa}\Delta t+\alpha\Delta t,\alpha\Delta t,\alpha\Delta t\right) . \tag{63}$$

After a and α are determined through moment matching, the neutron density will be calculated. One of the terms within the neutron density, as shown in (23), is the δk term. The δk term here is an integral which can be evaluated using exponential moment methods. The isolated integral is

$$n^{\delta\kappa}\left(t\right) = ae^{-\overline{\kappa}t}\int_0^t e^{\left(\overline{\kappa}+\alpha\right)t'}\delta\kappa\left(t'\right)dt' \tag{64}$$

.

Substituting in the polynomial source assumption and multiplying and dividing by Δt gives us

$$n^{\delta\kappa}\left(\Delta t\right) = a\Delta t\sum_{p=0}^{P_\rho}\delta\kappa_p e^{-\overline{\kappa}\Delta t}\int_0^t\left(\frac{t'}{\Delta t}\right)^p e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)\frac{t'}{\Delta t}}\frac{dt'}{\Delta t} \tag{65}$$

.

Apply the following change of variable:

$$u = \frac{t'}{\Delta t} \tag{66}$$

$$n^{\delta\kappa}\left(\Delta t\right) = a\Delta t\sum_{p=0}^{P_\rho}\delta\kappa_p e^{-\overline{\kappa}\Delta t}\int_0^1 u^p e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)u}du \tag{67}$$

and another change of variables:

$$v = 1-u \tag{68}$$

$$n^{\delta\kappa}\left(\Delta t\right) = a\Delta t\sum_{p=0}^{P_\rho}\delta\kappa_p e^{-\overline{\kappa}\Delta t}e^{\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)}\int_0^1\left(1-v\right)^p e^{-\left(\overline{\kappa}\Delta t+\alpha\Delta t\right)v}dv \tag{69}$$

Again, the form of the exponential moment function reveals itself:

$$n^{\delta\kappa}\left(\Delta t\right) = a\Delta t\sum_{p=0}^{P_\rho}\delta\kappa_p e^{\alpha\Delta t}\mathcal{M}_p\left(\overline{\kappa}\Delta t+\alpha\Delta t\right) \tag{70}$$

.

IV.D. Neutron Density Determination

As stated previously, the method seeks to find an approximation of the neutron density over a time step. Our two unknowns, a and α, require a

system of two equations in order to be determined. By matching the 0^{th} and 1^{st} moment of equations (21) and (23), the unknowns can be solved:

$$\begin{aligned} a\mathcal{M}_0\left(-\alpha\Delta t\right) = n\left(0\right)\mathcal{M}_0\left(\overline{\kappa}\Delta t\right) + \sum_i c_i\left(0\right)\left(\lambda_i\Delta t\right)\mathcal{M}_0\left(\overline{\kappa}\Delta t, \lambda_i\Delta t\right) \\ + \sum_{p=0}^{P_{source}} \frac{s_p\Delta t}{p+1}\mathcal{M}_{p+1}\left(\overline{\kappa}\Delta t\right) \\ + a\Delta t\sum_{p=0}^{P_\rho} \delta\kappa_p\mathcal{M}_p\left(-\overline{\kappa}\Delta t - \alpha\Delta t, -\alpha\Delta t\right) \\ + a\sum_i \beta_i\left(\lambda_i\Delta t\right)\left(\frac{\Delta t}{\Lambda}\right)\mathcal{M}_0\left(\overline{\kappa}\Delta t, \lambda_i\Delta t, -\alpha\Delta t\right). \end{aligned} \tag{71}$$

$$\begin{aligned} a\mathcal{M}_1\left(-\alpha\Delta t\right) = n\left(0\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t\right) + \sum_i c_i\left(0\right)\left(\lambda_i\Delta t\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t, \lambda_i\Delta t\right) \\ + \sum_{p=0}^{P_{source}} \frac{s_p\Delta t}{\left(p+1\right)\left(p+2\right)}\mathcal{M}_{p+2}\left(\overline{\kappa}\Delta t\right) \\ + a\Delta t\sum_{p=0}^{P_\rho} \delta\kappa_p\mathcal{M}_p\left(-\overline{\kappa}\Delta t - \alpha\Delta t, -\alpha\Delta t, -\alpha\Delta t\right) \\ + a\sum_i \beta_i\left(\lambda_i\Delta t\right)\left(\frac{\Delta t}{\Lambda}\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t, \lambda_i\Delta t, -\alpha\Delta t\right). \end{aligned} \tag{72}$$

Some simplification is useful for solving for our unknowns. The combination of variables $\alpha\Delta t$ is solved for using a rootsolver. Thus, the terms containing $\alpha\Delta t$ are collected on one side, which are all conveniently multiplied by a. For the 0^{th} moment equality:

$$aA_0\left(\alpha\Delta t\right) = B_0 \tag{73}$$

where

$$A_0\left(\alpha\Delta t\right) = \mathcal{M}_0\left(-\alpha\Delta t\right) - \Delta t\sum_{p=0}^{P_\rho}\delta\kappa_p\mathcal{M}_p\left(-\overline{\kappa}\Delta t - \alpha\Delta t, -\alpha\Delta t\right) \\ -\sum_i \beta_i\left(\lambda_i\Delta t\right)\left(\frac{\Delta t}{\Lambda}\right)\mathcal{M}_0\left(\overline{\kappa}\Delta t, \lambda_i\Delta t, -\alpha\Delta t\right) \tag{74}$$

and

$$B_0 \equiv n\left(0\right)\mathcal{M}_0\left(\overline{\kappa}\Delta t\right) + \sum_i c_i\left(0\right)\left(\lambda_i\Delta t\right)\mathcal{M}_0\left(\overline{\kappa}\Delta t, \lambda_i\Delta t\right) \\ + \sum_{p=0}^{P_{source}}\frac{s_p\Delta t}{p+1}\mathcal{M}_{p+1}\left(\overline{\kappa}\Delta t\right). \tag{75}$$

For the 1st moment:

$$aA_1\left(\alpha\Delta t\right) = B_1 \tag{76}$$

where

$$A_1\left(\alpha\Delta t\right) \equiv \mathcal{M}_1\left(-\alpha\Delta t\right) - \Delta t\sum_{p=0}^{P_\rho}\delta\kappa_p\mathcal{M}_p\left(-\overline{\kappa}\Delta t - \alpha\Delta t, -\alpha\Delta t, -\alpha\Delta t\right) \\ -\sum_i \beta_i\left(\lambda_i\Delta t\right)\left(\frac{\Delta t}{\Lambda}\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t, \lambda_i\Delta t, -\alpha\Delta t\right) \tag{77}$$

and

$$B_1 \equiv n\left(0\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t\right) + \sum_i c_i\left(0\right)\left(\lambda_i\Delta t\right)\mathcal{M}_1\left(\overline{\kappa}\Delta t, \lambda_i\Delta t\right) \\ + \sum_{p=0}^{P_{source}}\frac{s_p\Delta t}{\left(p+1\right)\left(p+2\right)}\mathcal{M}_{p+2}\left(\overline{\kappa}\Delta t\right). \tag{78}$$

By taking the ratio of these two equations, a is eliminated and a single equation and unknown remains that can be put through a rootsolver algorithm of choice.

$$\frac{A_1\left(\alpha\Delta t\right)}{A_0\left(\alpha\Delta t\right)} = \frac{B_1}{B_0} \tag{79}$$

$$f\left(\alpha\Delta t\right) \equiv B_0 A_1\left(\alpha\Delta t\right) - B_1 A_0\left(\alpha\Delta t\right) = 0 \tag{80}$$

The general behavior of this root-solving problem and our choice of algorithm is explained in V.A. Once α has been determined, a can be calculated as well using either the 0^{th} or 1^{st} moment:

$$a = \frac{B_0}{A_0\left(\alpha\Delta t\right)}. \tag{81}$$

Finally, the neutron and precursor densities can be evaluated at time Δt using equations (22) and (23). Substituting in a and α, and applying our changes of the integrals into moment functions one finds the following equations for the densities at the end of the time step:

$$c_i\left(\Delta t\right) = c_i\left(0\right)e^{-\lambda_i\Delta t} + a\beta_i\frac{\Delta t}{\Lambda}e^{\alpha\Delta t}\mathcal{M}_0\left(\alpha\Delta t + \lambda_i\Delta t\right) \tag{82}$$

$$\begin{aligned} n\left(\Delta t\right) = n\left(0\right)e^{-\overline{\kappa}\Delta t} + \sum_{p=0}^{P_{source}} s_p\Delta t\,\mathcal{M}_p\left(\overline{\kappa}\Delta t\right) + \sum_i \lambda_i\Delta t\,e^{-\lambda_i\Delta t} \times \\ \left[c_i\left(0\right)\mathcal{M}_0\left(\overline{\kappa}\Delta t - \lambda_i\Delta t\right) + a\beta_i\frac{\Delta t}{\Lambda}\mathcal{M}_0\left(\overline{\kappa}\Delta t - \lambda_i\Delta t, -\alpha\Delta t - \lambda_i\Delta t\right)\right] \\ + a\Delta t\sum_{p=0}^{P_\rho}\delta\kappa_p e^{\alpha\Delta t}\mathcal{M}_p\left(\overline{\kappa}\Delta t + \alpha\Delta t\right), \end{aligned} \tag{83}$$

This concludes the overview of the algorithm used to solve a PRKE system using exponential moment methods. The devices and methods chosen to realize this algorithm now need to be specified.

V. IMPLEMENTATION

The code created to calculate the neutron density utilizes various methods that were chosen or constructed to complete modular tasks within the problem in a robust manner. These tasks include root solving, domain shifting and adapting to various forms of reactivity. Additionally, planning the structure of the code carefully has allowed the flexibility of new tools to supplement the original foundation. A copy of the code can be found in Appendix I.A.

V.A. Root-Solving

Root-solving for $\alpha\Delta t$ can be done with many methods that vary in efficiency and reliability. Knowing a bracket that contains the root can initially accelerate the process using a bisection method. Then a rapidly converging method can be used. Rather than jumping in blind for the first iteration of root-solving, a quick search is done for a possible bracket with the assumption that the function is monotonically increasing or decreasing and defined everywhere. This assumption has been proven true for all cases used during testing, but may not actually be a property of the method. The value of function (80) is evaluated at 0. The slope is also found by evaluating the function at a small step forward and backwards from that point, which is determined by an input parameter. Using this information, the bracket can be found by approaching the root by doubling the

step size until the sign of the function changes. Once the sign changes, a bracket is found given by $f(0)$ and some other point that has the opposite sign of $f(0)$ which may exist above or below 0. The exception is for a steady state system where the neutron density does not change. In this case, $f(0)$ will be the root itself. Once the bracket is set around the root, an estimation of the root location is calculated by drawing a secant between the bracketed points. The function is evaluated at this point and will replace the bracketed point that shares the same sign. This is known as the method of false position in root-solving. It combines features of the secant method and bisection method to quickly converge on the point, but also guarantee a result given the function is well behaved. The code for the bracket search and secant/bisection rootfinder can be found in Appendix VIII.B.8 and Appendix VIII.B.9.

V.B. Solution to Sinusoidal Reactivity

The design of our algorithm requires the reactivity term to be in the form of a polynomial. However, as stated earlier, realistic problems will often have a functional form of reactivity that is not a polynomial. For example, control rods being inserted and withdrawn periodically on a wheel can create a sinusoidal form. This creates a need to fit a polynomial to a given form. There are several methods that fit a polynomial to other functional forms. The design choices must fulfill two major objectives: quickly fit a polynomial because the method

will have to be done for every time-step, and accurately fit a polynomial throughout the domain of interest because error from our fit will propagate into our final solution. A polynomial fit is needed within our time step for the following form of reactivity:

$$\rho(t) = a\sin(\theta + \omega t) \tag{84}$$

With these constants that define the sinusoid, a few options can be used to create a polynomial fit. First off, the smaller the time scale with respect to the period of the sinusoid, the better the fit will be for the same order of fit. However, this will require more steps. In general, the step size requirement of approximating the reactivity as an exponential will be sufficiently small for realistic sinusoidal problems, such as those caused by control rod movement.

Well-known methods, such as a Taylor series are useful for predicting function behavior near a specific point. Accuracy is required throughout a domain and thus a Taylor expansion will entail too many terms before the accuracy goal is satisfied. Instead, a type of moment matching using Legendre polynomials is applied. Various properties of Legendre polynomials allow us to create fits with some quick and simple arithmetic, and the behaviors of the original function are mimicked throughout the domain.

Fitting a polynomial to a given function using Legendre polynomials is simple and systematic, as long as the function behaves somewhat like a polynomial in

the region of interest. The polynomial fit requires the Legendre polynomials up to the order of our fit. The first five Legendre polynomials were shown to be overly sufficient.

Table 1: First Four Legendre Polynomials

n	$P_n(x)$
0	1
1	x
2	$\frac{1}{2}\left(3x^2-1\right)$
3	$\frac{1}{2}\left(5x^3-3x\right)$
4	$\frac{1}{8}\left(35x^4-30x^2+3\right)$

The sum is taken of the inner product of our function of interest with each Legendre polynomial up to the order desired to fit the function. The formula for this summation is

$$f_{fit}(x)=\sum_{n=0}^{order}P_n(x)\left(\frac{2n+1}{2}\right)\int_{-1}^{1}f(x')P_n(x')dx' \quad \text{for } -1\le x\le 1\,. \tag{85}$$

Although this fit is a simple calculation, the function can have many forms and the nature of the inner product may require special treatment. Creating an

algorithm to solve the inner product in a robust manner for any function and any order can be a feat on its own. Instead, knowledge of the function required allows the creation of a reference table of solutions. In the proposed situation, the function is sinusoidal, and only a specified number of inner products are required which is determined by user input.

The first step in solving for the table of inner products is shifting the function into the correct domain. Legendre polynomials are orthogonal between -1 and 1 and thus require us to shift the sinusoid into the same domain, -1 to 1. For the general sinusoid given in (84), the three constants are shifted as

$$\begin{aligned} a_{new} &= a \\ \theta_{new} &= \theta + \frac{\omega \Delta t}{2} + \omega t_i \\ \omega_{new} &= \frac{\omega \Delta t}{2} \end{aligned} \tag{86}$$

where t_i is the time at the beginning of the step, and Δt is the length of the step. The order of our polynomial is limited to 4^{th} order. The coefficients for these Legendre polynomials are given on the next page.

Table 2: Coefficients of Legendre Polynomials

Polynomial Number	Order	Coefficient
0	0	1
1	1	1
2	0	-.5
2	2	1.5
3	1	-1.5
3	3	2.5
4	0	.125
4	2	3.75
4	4	4.375

These coefficients are reconventionalized in the form $p(n,order)$. For example, $p(3,1) = -1.5$. Additionally, the inner product of our function in (84) and each Legendre polynomial is required. The solutions for the first four of these inner products are given as

$$
\begin{aligned}
cip(0) &= \frac{a_{new}\sin(\theta_{new})\sin(\omega_{new})}{\omega_{new}} \\
cip(1) &= a_{new}\left[\frac{\cos(\theta_{new})\sin(\omega_{new})}{\omega_{new}^{2}} - \frac{\cos(\theta_{new})\cos(\omega_{new})}{\omega_{new}}\right] \\
cip(2) &= \rho_{1_n}\left[\frac{3\cos(\omega_{new})\sin(\theta_{new})}{\omega_{new}^{2}} - \frac{3\sin(\theta_{new})\sin(\omega_{new})}{\omega_{new}^{3}} + \frac{\sin(\theta_{new})\sin(\omega_{new})}{\omega_{new}}\right] \\
cip(3) &= \rho_{1_n}\left[\begin{array}{l}\frac{15\cos(\theta_{new})\cos(\omega_{new})}{\omega_{new}^{3}} - \frac{\cos(\theta_{new})\cos(\omega_{new})}{\omega_{new}} \\ -\frac{15\cos(\theta_{new})\sin(\omega_{new})}{\omega_{new}^{4}} + \frac{6\cos(\theta_{new})\sin(\omega_{new})}{\omega_{new}^{2}}\end{array}\right] \\
cip(4) &= \rho_{1_n}\left[\begin{array}{l}\frac{-105\cos(\omega_{new})\sin(\theta_{new})}{\omega_{new}^{4}} + \frac{10\cos(\omega_{new})\sin(\theta_{new})}{\omega_{new}^{2}} \\ +\frac{105\sin(\theta_{new})\sin(\omega_{new})}{\omega_{new}^{5}} - \frac{45\sin(\theta_{new})\sin(\omega_{new})}{\omega_{new}^{3}} \\ +\frac{\sin(\theta_{new})\sin(\omega_{new})}{\omega_{new}}\end{array}\right]
\end{aligned}
\tag{87}
$$

Given these inner product constants and the Legendre polynomial coefficients, the fitted polynomial is solved through a mathematics package and can be calculated with simple arithmetic.

For a 0^{th} order fit:

$$\rho_{0poly} = cip(0)p(0,0) \tag{88}$$

.

For a 1^{st} order fit:

$$\rho_{1poly} = 3cip(1)p(1,1)t \tag{89}$$

.

For a 2^{nd} order fit:

$$\rho_{2poly} = cip(0)p(0,0) + 5cip(2)p(2,0) + 5cip(2)p(2,2)t^2 \tag{90}$$

.

For a 3rd order fit:

$$\rho_{3poly} = 3cip(1)p(1,1)t + 7cip(3)p(3,1)t + 7cip(3)p(3,3)t^3 \quad . \tag{91}$$

And finally a 4th order fit:

$$\begin{aligned}\rho_{4poly} = {} & cip(0)p(0,0) + 5cip(2)p(2,0) + 9cip(4)p(4,0) \\ & 5cip(2)p(2,2)t^2 + 9cip(4)p(4,2)t^2 + 9cip(4)p(4,4)t^4 \quad .\end{aligned} \tag{92}$$

Once the polynomial order is chosen, a fit for the sinusoid is now available. All that remains is to move the polynomial from the -1 to 1 domain back into the t_i to $t_i + \Delta t$ domain. Binomial coefficients are used to shift polynomials from one domain to another and the method in which this is done so is explained thoroughly in the next section.

V.C. Initial Condition Domain Shift

After each time step, recalculating the average reactivity is required before repeating the algorithm. The problem has a few nuances that must be considered after the first time step in order to correctly calculate future iterations. Each time step, the average reactivity is calculated and is a major part of our algorithm's calculation. Additionally, our derivation was for the first time step, and thus the initial time is 0, greatly simplifying a lot of the math. Rather than deriving everything once again for the time between t and $t + \Delta t$, the final population densities of the previous iteration are used as the initial condition for

the next iteration and the reactivity and source polynomials are shifted by Δt. The reactivity and source are both assumed to be polynomials and thus the method applied to both is identical. For a polynomial source of the form

$$S(t) = \sum_{p=0}^{P_{source}} S_p t^p \tag{93}$$

one can apply a shift of some amount of time denoted as t_{passed}. Thus the shifted source is now

$$S_{shifted}(t) = \sum_{p=0}^{P_{source}} S_p \left(t + t_{passed}\right)^p . \tag{94}$$

Once again, binomial expansion is used to determine the new coefficients for the source:

$$S_{p_{new}} = \sum_{j=p}^{P_{source}} \binom{j}{p} S_j \left(t_{passed}\right)^{j-p} \tag{95}$$

and now a revised shifted source equation can be used.

$$S_{shifted}(t) = \sum_{p=0}^{P_{source}} S_{p_{new}} t^p \tag{96}$$

To reiterate, reactivity is done the same way as if it is naturally in a polynomial form.

For a sinusoidal reactivity, this shift must occur before refitting a polynomial every time-step. For this form shown in (84), we can shift by t_{passed}. In order to

do so, the constants ρ_1 and ρ_3 remain the same, but the second constant must shift accordingly:

$$\rho_{2new} = \rho_2 + \rho_3 t_{passed} \tag{97}$$

$$\rho_{shifted}(t) = \rho_1 \sin(\rho_{2new} + \rho_3 t) \tag{98}$$

After shifting the reactivity, the polynomial fit technique shown in V.B can be used to meet the requirements of our algorithm. Each polynomial fit is only good for the time domain it was calculated, so our problem requires a constant shifting of the reactivity.

V.D. Error Control Algorithm with Adaptive Time Steps

The purpose of error control is to minimize the costs of finding a solution while keeping the error below some stated tolerance. In order to design such a feature into our algorithm, one must first understand the source and behavior of error in this particular problem. Error introduces itself in our solution from several sources. Our limitation of precision, the number of digits stored for calculation purposes, results in some small finite amount of error. Another source of error is created due to the fact that we the method solves an approximation of the problem. Various sources of error will usually differ by orders of magnitude in relevance. While designing an algorithm, knowing which source of error is the most relevant and creating a method to reduce it is the

main mechanism for improving the fidelity of the solution. From an error control perspective, if one source of error clearly dominates, the other sources may be ignored.

Error control schemes in general operate by solving an iteration of a given problem and comparing the error to a specified tolerance. The tolerance will determine if the solution is usable and if the step size was optimal. Depending on this result, the scheme may redo the iteration with a different time step, use a different time step on the next iteration, or use the same time step on the next iteration. In order for this decision to be made, the error must be known to compare to the tolerance.

Due to the nature of our problem, it is very difficult to know exactly what the exact solution is and thus the exact error. For verification purposes, the solution from a mathematics package is produced and compared, a luxury that won't be incorporated with the code on a regular basis. However, using the information from the verification portion of the results, one can gain some insight of the properties of the error. When tolerances are set very tight and the time steps taken are small relative to all the inputs, one may notice that the error associated with our solution is on the same order of magnitude as the precision allowed. This result is expected and additional reduction of the tolerances or step size does not further improve the fidelity of the answer. However, increasing the step size reveals another source of error that is associated with the properties of our

algorithm. Once this source of error dominates over the precision error, regression can be used to find the form of the error. Our results imply that this source of error follows the following form:

$$\varepsilon = a(\Delta t)^p \tag{99}$$

where ε is the relative error and a and p are simply constants that are determined by the nature of the problem being solved. Using this information, an adaptive error control scheme can be created that changes the step size based on an estimation of the error. The error can be estimated by calculating the solution twice using different sized time steps. For our algorithm specifically, the step size is changed by a factor of two. This design decision doesn't allow much flexibility in how close to the optimal step size one can get and may seem crude. However, there are two main redeeming features of this choice. First, only three calculations must be done in order to estimate the error, solving the problem twice in succession with the half step size, and once with the full step size. Second, if the error proves to be minimal, the solution of the error control now becomes the solution to the main problem, thus minimizing the number of calculations necessary.

Although the actual solution of a specified problem is not known, one can presume that the error, our solution and the real solution are related in the following way:

$$\varepsilon = a(\Delta t)^p = \frac{\tilde{S}(\Delta t) - S}{S}. \tag{100}$$

where S is the actual solution and $\tilde{S}(\Delta t)$ is our solution for given step size Δt (not to be mistaken with $S(t)$, the source term). The solution can then be found in terms of other factors.

$$\tilde{S}(\Delta t) = S\Big(1 + a(\Delta t)^p\Big) \tag{101}$$

If one assumes the same amount of error is accumulated when two half steps are added, then the associated error for the half solution will be

$$\tilde{S}'(\Delta t_h) = \tilde{S}(\Delta t_h) + \tilde{S}(\Delta t_h) - S = S\Big(1 + 2a(\Delta t_h)^p\Big). \tag{102}$$

Solving the same problem again using full step sizes yields

$$\tilde{S}(2\Delta t_h) = S\Big(1 + 2^p a(\Delta t_h)^p\Big). \tag{103}$$

The two results are then combined to eliminate the actual solution, which is unknown.

$$\frac{\tilde{S}(2\Delta t_h) - \tilde{S}'(\Delta t_h)}{\tilde{S}'(\Delta t_h)} = \frac{S\Big(2^p a(\Delta t_h)^p - 2a(\Delta t_h)^p\Big)}{S\Big(1 + 2a(\Delta t_h)^p\Big)} \tag{104}$$

If the magnitude of the step size is relatively small to the solution, one can use the following approximation:

$$\frac{\tilde{S}(2\Delta t_h) - \tilde{S}(\Delta t_h)}{\tilde{S}(\Delta t_h)} \approx (2^p - 2)a(\Delta t_h)^p. \tag{105}$$

And finally, this can be used as a rough estimate of the error.

$$\varepsilon = a(\Delta t_h)^p \approx \frac{\tilde{S}(2\Delta t_h) - \tilde{S}(\Delta t_h)}{(2^p - 2)\,\tilde{S}(\Delta t_h)} \tag{106}$$

With this information, a pair of tolerances can be used to manipulate the behavior of our algorithm by changing step size. If the relative error estimation is below some tolerance, the doubling tolerance, the information found by our error scheme is assumed to be of an acceptable error level and is justifiably recorded. Next, the size of the step is doubled because the above derivation proposes that a larger step size would also be acceptable and require less total calculations in the long run. However, the next iteration of error control will test to see if this assumption is correct. If the relative error estimation is above some other critical tolerance, the halving tolerance, the error estimation has become too great to risk using the information that was just computed. Instead, the step size will be halved, and the error control routine will test the new value. After some finite number of iterations, the error control will find the step size that contributes an acceptable amount of error or fail to reach a value and deliver a stop command. If the relative error estimation is between our two tolerances, then the algorithm assumes the answer has enough fidelity to be acceptable, but not enough to warrant the search of a more effective step size. Thus, the total error accumulated will be some magnitude between the thresholds per step. However, it is known that error accumulates, so for long problems that may take

many steps to solve, the tolerances should be set such that the accumulation of error meets the prespecified goals.

The behavior and performance of our error control scheme is attributed partially to the tolerances set. The halving tolerance is the main mechanism for reducing the error. Due to the fact that a given result won't be accepted and recorded unless the halving tolerance is met, tightening up the halving tolerance will result in less accumulation of error in the long run in exchange for shorter step sizes, and thus increasing computational cost. The doubling tolerance has an interesting effect on the accuracy and cost of operation. If the doubling tolerance is set too strict, the step size may never double into another acceptable step size, increasing computational cost by a factor of 2. With an extremely strict doubling tolerance, the increase in operation cost grows by a factor of 2^n where n is the number of times it could have doubled and still remained under the halving threshold. In return, the accumulation of error will be lower than one would expect from the halving tolerance. However, if the doubling tolerance is set too loose (very close to the halving tolerance), the additional error may cause the next iteration to trigger the halving mechanism. That is, the step size is never in the comfortable region between the two tolerances. In this case, the step size will jump back and forth and half of the iterations will not be used. The result is that the computational cost increases by a factor of 2.

Knowing these properties of this error control scheme, one may note that even with well set tolerances, it is possible that the algorithm proposed step size may be suboptimal by up to a factor of 2. This fact is one of the attributes that makes this error control scheme somewhat crude. With more information on the effects of step size on error, the algorithm can be enhanced to set the step size accordingly. This information can only be provided with extensive testing, which is not within the scope of our goals.

Finally, an additional method of error control has been developed during our research. Rather than using a raw calculation with assumptions about how error is a function of step size, one can calculate our densities to a higher order using another Picard iteration. This method may provide an advantage in situations where there is some source of error that does not follow the standard form for error associated with the step size. Situations like these are not observed in our results, but perhaps another functional form of reactivity can produce such a circumstance. This method was not pursued because the original error control scheme was successful for the selected test cases, but a general introduction to using Picard iterations is provided in Appendix VIII.A.

VI. TESTING, RESULTS AND ANALYSIS

Our testing procedure aims to meet the goals listed in the problem statement and goals section. These goals generally fit within two categories: validation and performance assessment. These categories are appropriate because numerical methods in general are judged based on the accuracy of the result they produce, and the number of operations it took to reach the result. Unfortunately, improvements in one of these properties usually results in reduction of the other. For validation based goals, the value of our method can be quantified based on the error within the result produced. For performance assessment, the cost of running our method can be quantified based on the number of operations necessary to complete it. This is done by exploring the relationship between step size and relative error, knowing that step size is the main factor that determines computational costs. In essence, our testing will demonstrate the relationship between accuracy and cost for our particular method.

In order to quantify our success in meeting our goals, the correct solution to each test problem is required. Mathematica, a standard math package, is used in order to solve the same test problems using the built in numerical differential equation solver. For certain test problems, the code did not converge to the result produced by Mathematica using default options. This implies that either there is an error in the method, or some error accumulated in the method used

by Mathematica that exceeds the precision required. By increasing the working precision, precision goal, and accuracy goal on Mathematica, this issue can be rectified for all of our selected test problems. The NDSolve() function is used to solve the PRKEs which applies a combination of numerical methods including *Runge-Kutta*, *trapezoid rule,* and *extrapolation.* The performance effecting options were set arbitrarily high to ensure accuracy at values of 30 for the working precision, 15 for the precision goal and 15 for the accuracy goal. For the purpose of our testing, the results given by Mathematica are assumed to be correct to at least as many digits as the precision level set within our code. The worksheet for our Mathematica testing suite can be found in Appendix VIII.C.

The test problems throughout these results use a six group approximation for the production of all results produced within this paper. The β_i and λ_i values are given within the following table.

Table 3: Values for Six Group Approximation

Group Number	β_i	λ_i
1	0.00021	0.0126
2	0.00142	0.0301
3	0.00127	0.112
4	0.00257	0.301
5	0.00075	1.14
6	0.00027	3.01

The neutron lifetime used for the majority of our tests, $\Lambda = 0.0824528053$ s, is based on an experimental reactor. However, a longer lifetime like this does not accurately represent the average reactor once it goes through prompt criticality. In this situation, the behavior of the reactor is no longer dominated by delayed neutrons and our choice of prompt lifetime will affect the results greatly. For the prompt criticality tests, a neutron lifetime of $\Lambda = 3 \times 10^{-5}$ s is used instead which redefines the reactor conditions. The fact that all of the tests do not use this lifetime is simply a result of observations found while testing our method with prompt criticality, which was final task performed during research. The original longer lifetime is sufficient for showing the behaviors in situations where delayed neutrons determine the behavior of the population densities.

VI.A. Solution: Trivial Steady State Conditions

Due to the fact that most reactors are usually left in a steady state condition or naturally enter into one eventually, creating a set of steady state conditions and confirming that our algorithm produces steady state results is a good starting point for our validation process. One can use any set of neutron and precursor concentrations, so our test cases arbitrarily use a neutron density of 1×10^{8}. We can now solve for the precursor densities that match. One may note that our densities can be of any unit and our algorithm is correct as long as all densities have the same unit.

There are two steady state conditions. The first is the steady state condition of criticality. With no source inserted into the reactor and $\rho = 0$, the reactor will eventually reach a steady state condition. This condition can be solved using the given information and equation (2) and (3).

$$0 = \left(\frac{0-\beta}{\Lambda}\right) n(t) + \sum_i \lambda_i c_i(t) \tag{107}$$

$$0 + \lambda_i c_i(t) = \frac{\beta_i}{\Lambda} n(t) \tag{108}$$

Given the neutron density set arbitrarily for test cases, one can use the second equation to solve for each of the precursor densities.

$$c_i(t) = \frac{\beta_i}{\Lambda \lambda_i} n(t) \tag{109}$$

Table 4: List of Neutron and Precursor Densities for Critical Steady State Conditions

n	100000000.000000
c_1	20213583.5233573
c_2	57215857.6807323
c_3	13752455.9328556
c_4	10355264.3830621
c_5	797904.612764104
c_6	108790.715308435

One can now use these initial conditions and verify our algorithms ability to correctly treat this steady state condition. Table 5 displays the parameters used within our algorithm and the results produced.

Table 5: Code Parameters Used for Critical Steady State Test

Δt	0.01
Number of Steps	100
$\rho(t)$	0
$S(t)$	0

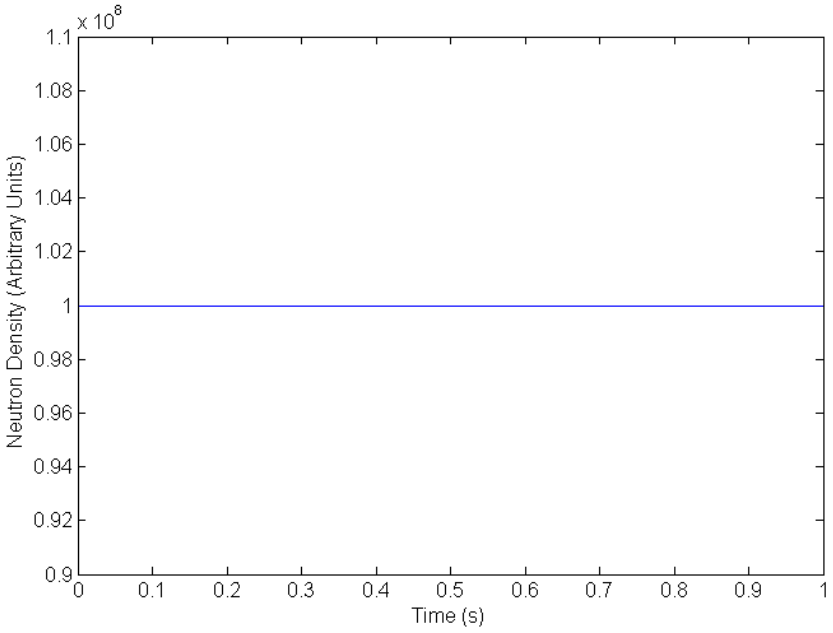

Figure 1: Critical Steady State Test Results

As expected, there is no change shown in the neutron densities. The precursor densities produce a similar result. Our algorithm uses the solution of each time step as the initial condition for the next step which is why the result continues with time.

As previously stated, there are two steady state conditions. If there is a source within the reactor and the reactor is subcritical, the densities will settle into densities that will allow the loss of neutrons to be offset by the addition of the source term. For this situation, the PRKEs are now

$$0 = \left(\frac{\rho_0 - \beta}{\Lambda}\right) n(t) + \sum_i \lambda_i c_i(t) + S_0 \tag{110}$$

$$0 + \lambda_i c_i(t) = \frac{\beta_i}{\Lambda} n(t)\,. \tag{111}$$

Once again, one can solve for the steady state conditions given a set neutron concentration. The same neutron population as before is used for simplicity. The source rate is now required in addition to the precursor densities. The source rate can be found by substituting (111) into (110).

$$S_0 = \frac{-n(t)\rho_0}{\Lambda} \tag{112}$$

The precursor densities are the same as the critical steady state conditions, but they can be written in terms of the source term for convenience.

$$c_i(t) = \frac{\beta_i S_0}{-\rho_0 \lambda_i} \tag{113}$$

The densities listed in Table 4 are used once again. The code parameters used are listed in Table 6.

Table 6: Code Parameters Used for Subcritical Steady State with Source Test

Δt	0.01
Number of Steps	100
$\rho(t)$	-0.001
$S(t)$	1212815.01140145

The results produced by these conditions are shown in Figure 2.

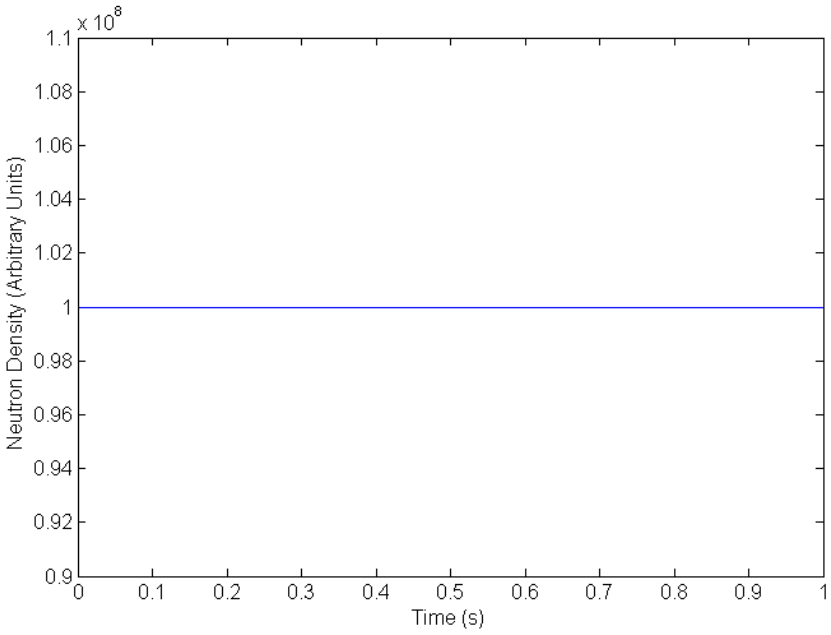

Figure 2: Subcritical Steady State with Source Test Results

Again, the solution is as expected. Our algorithm maintains the densities at the same value when steady state conditions are met.

VI.B. Verification: Linear Reactivity

Much of our error analysis can be based on a simple, but effective set of problem conditions. A reactor is assumed to be in critical steady state condition with the given densities shown in Table 4. At time $t = 0$, the reactivity step changes to some positive value. The reactivity then increases linearly. The step size is chosen to be 0.001. This value is approximately a sixth of our β value. This decision was to allow the delayed neutrons to dominate the behavior of the population, while still being critical. One may find similar conditions when powering up a reactor. A sample problem is created that fits this description and verify our algorithms ability to produce the correct answer. This sample problem can then be used later in our results to study the effects of our tolerances, step size, and error control. The code parameters are given in Table 7.

Table 7: Code Parameters Used for Linear Reactivity Test

Δt	0.001
Number of Steps	10000
$\rho(t)$	$0.001 + 0.00001t$
$S(t)$	0

The results produced from these conditions can be found in Figure 3.

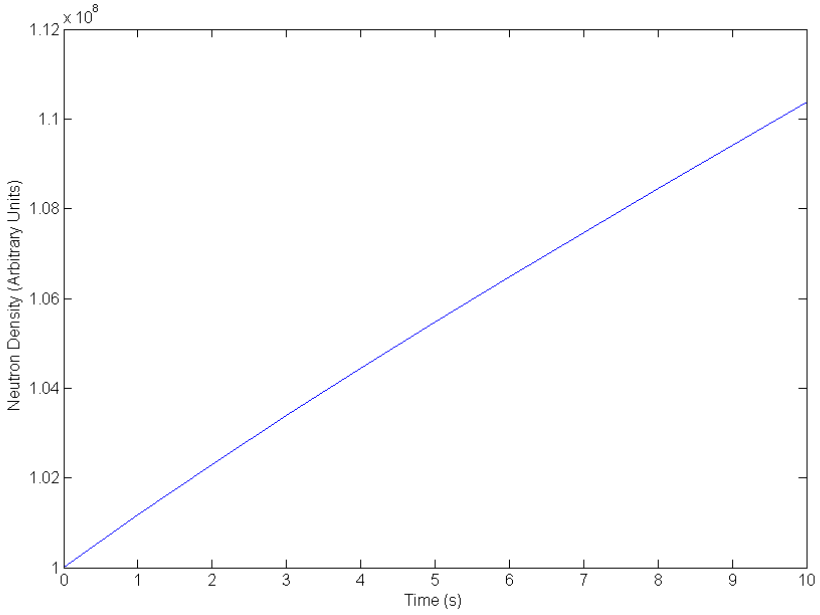

Figure 3: Linear Reactivity Test Results

The results seem to correctly fit the given conditions, but our goals require the analsis of the error of the neutron and precursor population densities. In order to do so, 11 points are sampled that are evenly spaced within our results

and the relative error is calculated against the result in Mathematica. For the precursor populations, the error accumulated is of the same order of magnitude for all groups, so the relative error of the precursor group with the largest λ_i value is arbitrarily exhibited to represent the behavior of the error in all of the precursor groups. The resulting analysis is shown in Figure 4.

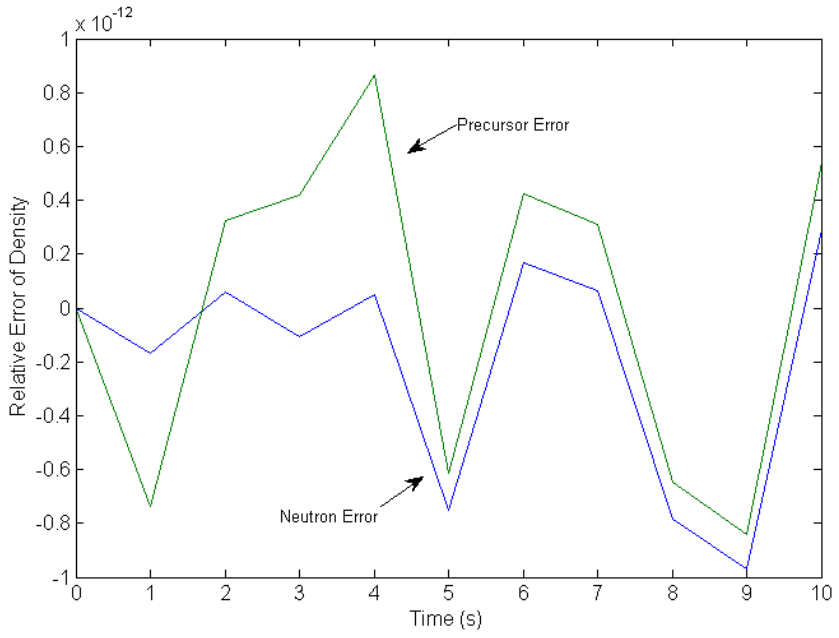

Figure 4: Error Development within Linear Reactivity Test

One may note that the error for both the neutron and precursor density are extremely small values, some positive, some negative, and initially with no pattern. Error of this magnitude matches what would be expected for minor precision loss caused by a single pass through our algorithm. The random positive and negative values suggest the same. Precision error such as this tends

to not accumulate like other errors because the values average out to 0. Instead, whatever error is accumulated is explained as a random walk. After some thousands of time steps, it seems like the neutron and precursor errors behaviors start to mimic each other. This may indicate the accumulation of error from a source that affects all densities equally. Error accumulation will be studied in our next result.

VI.C. Error Accumulation: Linear Reactivity

Our previous study of linear reactivity may hint at another source of error within our algorithm, which is expected. In order to differentiate this error from the precision error, the same conditions are used for a much longer time period to allow for the error to accumulate to a recognizable and quantifiable amount. The results imply that this accumulation of error is created by the fact that our method is solving an approximation of the actual problem. If this statement is true, the error accumulation rate would be related to the size of the steps taken. The same case can be solved with different time steps in order to view their effects on the accumulation of error. A solution utilizing small step sizes is first shown and the step size increases from there. The following parameters are used in the initial test.

Table 8: Code Parameters Used for Linear Reactivity

Error Accumulation Test 1

Δt	0.01
Number of Steps	100000
$\rho(t)$	$0.001 + 0.00001t$
$S(t)$	0

The neutron density produced by these conditions is shown in the next figure. Note that this is just an extension of Figure 3 by 990 seconds.

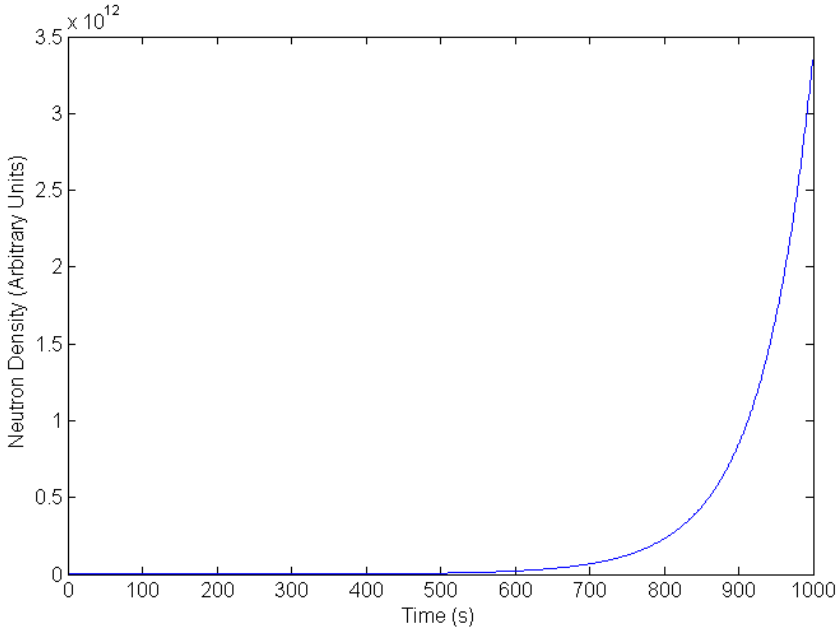

Figure 5: Extended Linear Reactivity Test Results

The actual interest of this test case is the behavior of the error. As with the previous case, one can view the relative error of the neutron density and a chosen precursor density in Figure 6.

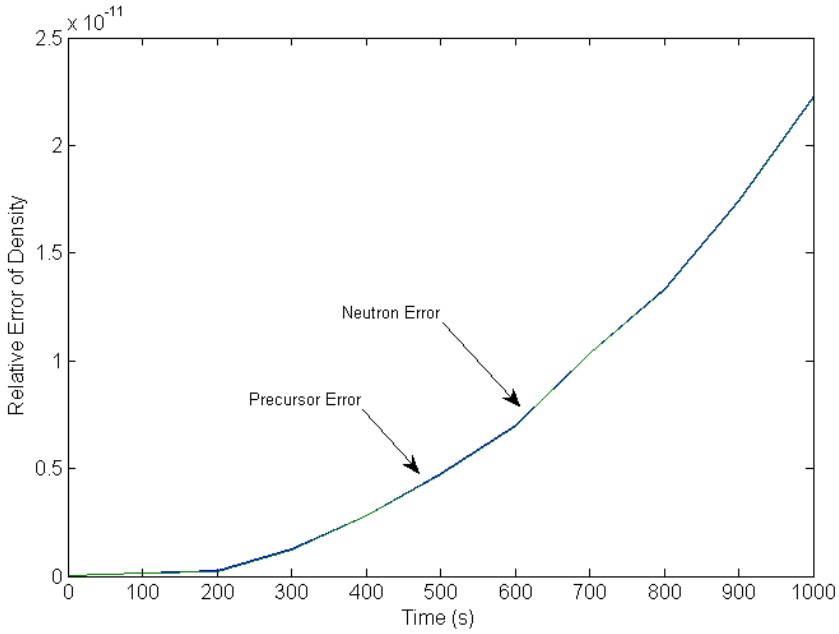

Figure 6: Error Development within Linear Reactivity Test 1

The neutron and precursor errors clearly follow each other, implying that their errors stem from the same source and they seem to have approximately the same relative magnitude. The overall error is still very low but clearly a couple digits of precision are now gone. One may note that within the first 200 seconds, the magnitude of the error is approximately the same as that found in our previous test overall. At some point, the accumulating error dominates over the

precision loss, and the error accumulates at an increasing rate. This fact may explain the curvature of our results. The random walk caused by precision loss is always present, but as time passes, the error from this source slowly becomes less relevant compared to systematic error accumulation from our algorithm. The error accumulated is all positive, that is the algorithm result is consistently higher than it should be. This behavior is a function of the conditions of the case itself. For negative reactivity, our algorithm tends to produce negative error, or results that are consistently lower than they should be. Rather than setting up an example showing this behavior, our study of sinusoids does an excellent job of characterizing this phenomenon.

As stated, the test setup seeks to verify that the error observed is a function of step size. The next test uses a step size that is 10 times larger than the previous test with the same conditions. This operation is repeated a few times in order to verify the dependence on step size.

Table 9: Code Parameters Used for Linear Reactivity

Error Accumulation Test 2

Δt	0.1
Number of Steps	10000
$\rho(t)$	$0.001 + 0.00001t$
$S(t)$	0

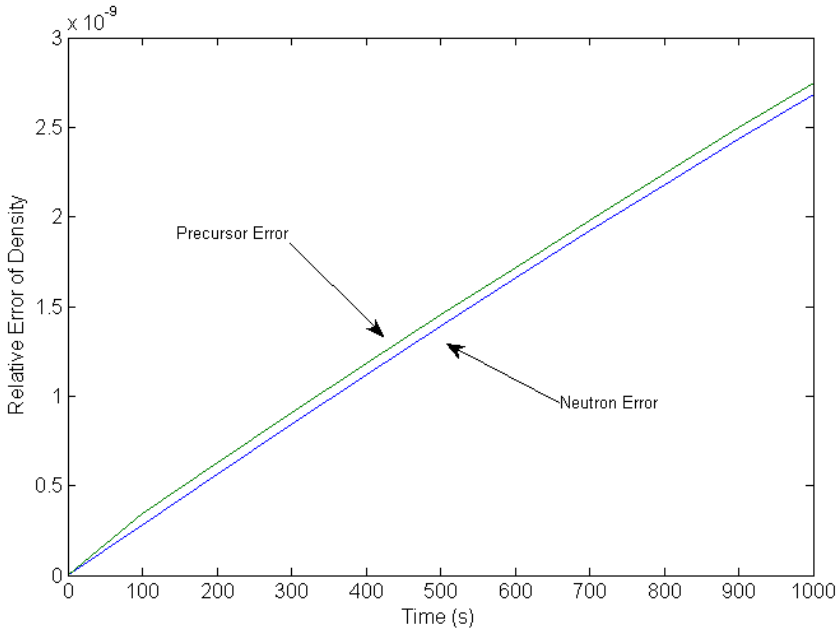

Figure 7: Error Development within Linear Reactivity Test 2

With a simple increase of time step, one can obtain a better understanding of this error source. The error has increased by multiple orders of magnitude. Additionally, even though the precursor and neutron density errors follow the same path, the precursor error is consistently greater than the neutron error by approximately the same amount. This is probably an artifact of our algorithm itself. Now that precision loss is many orders of magnitude below the error loss caused by our algorithm, the error development appears linear. That is, every step contributes an approximately set amount of error. One may predict that increasing the step size will reinforce these results further, unless some other mode of error becomes relevant. The rest of our tests are shown below.

Table 10: Code Parameters Used for Linear Reactivity

Error Accumulation Test 3

Δt	1
Number of Steps	1000
$\rho(t)$	$0.001 + 0.00001t$
$S(t)$	0

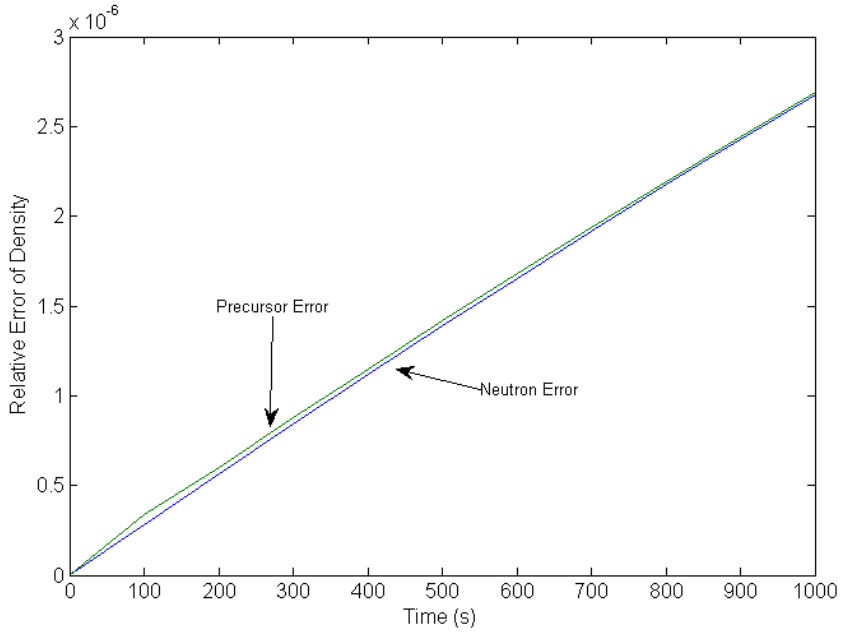

Figure 8: Error Development within Linear Reactivity Test 3

Table 11: Code Parameters Used for Linear Reactivity

Error Accumulation Test 4

Δt	10
Number of Steps	100
$\rho(t)$	$0.001 + 0.00001t$
$S(t)$	0

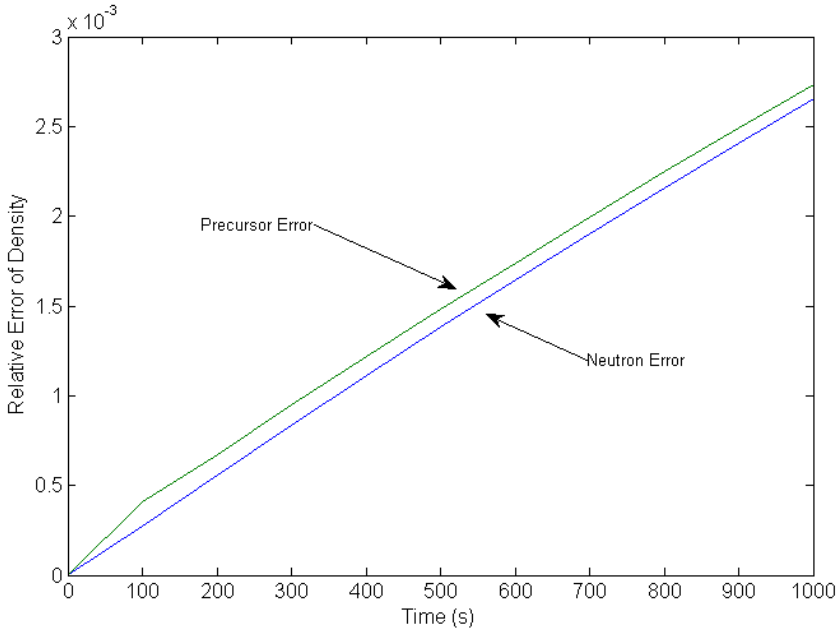

Figure 9: Error Development within Linear Reactivity Test 4

As expected these graphs are nearly identical, except each increase of step size results in a large increase in error. One additional nuance is shown in Figure 8.

The precursor and neutron densities slowly converge. Perhaps in the other test cases this behavior would eventually be observed given more time. However, the fact that it is only observed with 1 second time steps, but not with the .1 or 10 second time steps used in Figure 7 and Figure 9 does not give sufficient information to imply any logical pattern.

VI.D. Verification and Error Accumulation: Sinusoidal Reactivity

A sine wave was chosen to represent our ability to treat reactivity cases that are not easily represented by a polynomial. The algorithm fits a polynomial to a piece of the sine wave that is much smaller than the period using moment matching and then goes through the standard procedure. With this in mind, one can assume that in addition to the normal errors associated with the basic algorithm, our sinusoid fitting scheme will contribute some amount of error as well. By observing some test results involving sinusoidal reactivity, one can determine if this amount of additional error is relevant compared to the other sources of error.

First, a sine wave must be designed for this exploration. The amplitude of the sine wave should be a fraction of the β value. The choice used in the linear tests will work here as well. Additionally, a period must be picked for the sine wave. This sine wave is supposed to mimic a control rod being inserted and withdrawn from the reactor periodically. The shorter the period, the faster the

sine wave will turn, which puts more stress on the fitting algorithm. A quick period that is still somewhat realistic is somewhere around 1 second. Finally, the polynomial approximation of the sine is set to be 0^{th} order in order to observe the problem case before analyzing the effects of our fitting parameters. The problem design uses the same critical steady state, and at $t = 0$, the reactivity will follow the given sine wave.

Table 12: Code Parameters Used for Sinusoidal Reactivity Test 2 Periods 0^{th} Order

Δt	0.01
Number of Steps	200
$\rho(t)$	$0.001\sin(6.28t)$
$S(t)$	0
Poly. Order	0

The result produced by these conditions is shown in Figure 10.

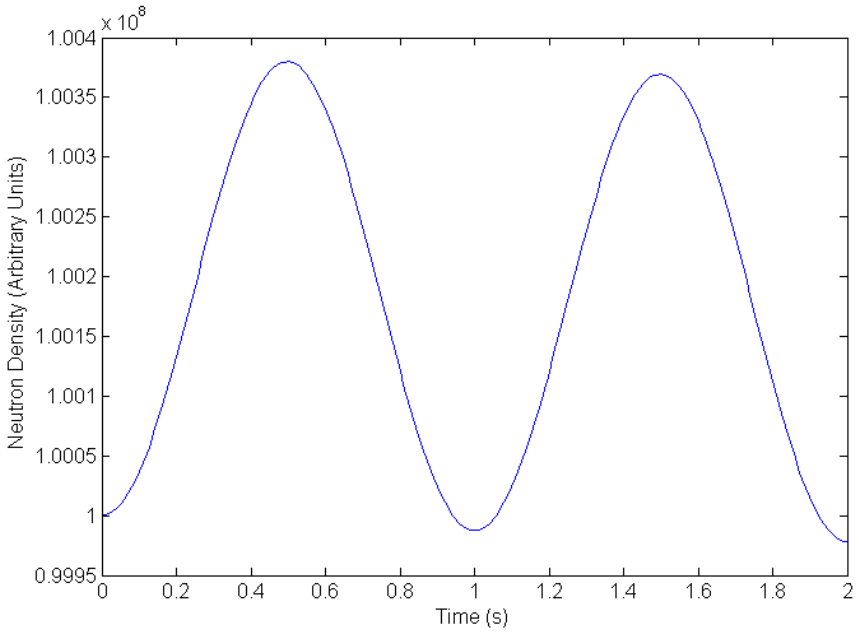

Figure 10: Sinusoidal Reactivity Test

Results 2 Periods 0^{th} Order

The result is also sinusoidal in nature, but offset from the reactivity by approximately $\pi / 2$. If one observes the plot closely, one may note that the second period is lower than the first in both peak and trough. This is the interaction that the delayed groups have on the densities for a sinusoidal reactivity. In order to better understand this interaction, the length of the same calculation is extended to 50 periods. Before discussion of the currect case is abandoned, additional insight can be gained from analyzing the error within the period of the sine. The discussion of accumulating effects over many periods is reconvened in VI.E.

As stated before, the sinusoidal reactivity has an additional component of error due to the polynomial approximation. In addition, a few new properties are introduced. There is curvature to our reactivity, and periodically our reactivity goes negative. These are all factors that will influence the sources of error in our solution. By observing relative error over time, perhaps some insight about these factors can be found. For the case listed in Table 12, the following error figure is produced.

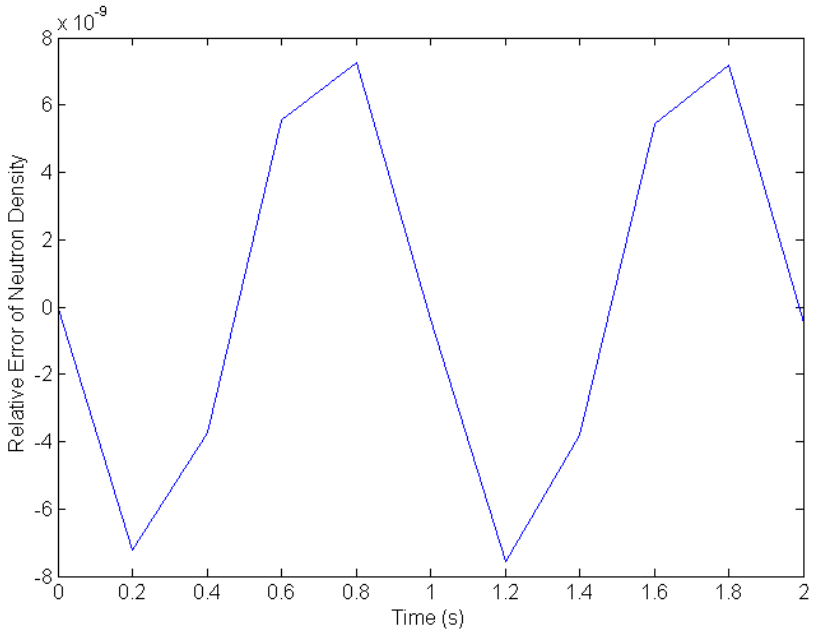

Figure 11: Neutron Error Development within Sinusoidal Reactivity Test 2 Periods 0^{th} Order

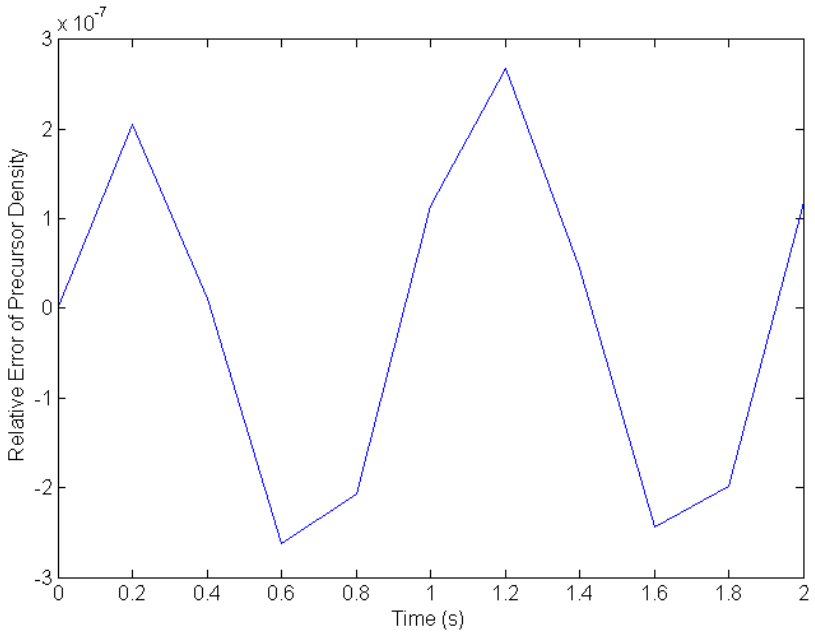

Figure 12: Precursor Error Development within Sinusoidal Reactivity Test 2 Periods 0^{th} Order

Our crude 11 point sampling has hurt the resolution of our figures now that curvature is involved, but information can still be gained by analyzing the plots. One initial observation is the order of magnitude of error between the neutron density and the precursor density is vastly different by a couple orders of magnitude. This differs from the linear case where the two had approximately the same amount of error. Additionally, both curves also have sinusoidal shapes in general, but they are out of synchronization. In order to determine the cause of the difference in magnitude of error, one can vary the problem parameters until the neutron and precursor densities reach a similar magnitude of error. By

doing this, it was found that 0^{th} order polynomial approximation of the sine contributed to the error of the precursors more than the neutron population. However, the reason this behavior occurs is still elusive. This phenomenon can be shown by running the same test with a better approximation.

Table 13: Code Parameters Used for Sinusoidal Reactivity Test 2 Periods 1^{st} Order

Δt	0.01
Number of Steps	200
$\rho(t)$	$0.001\sin(6.28t)$
$S(t)$	0
Poly. Order	1

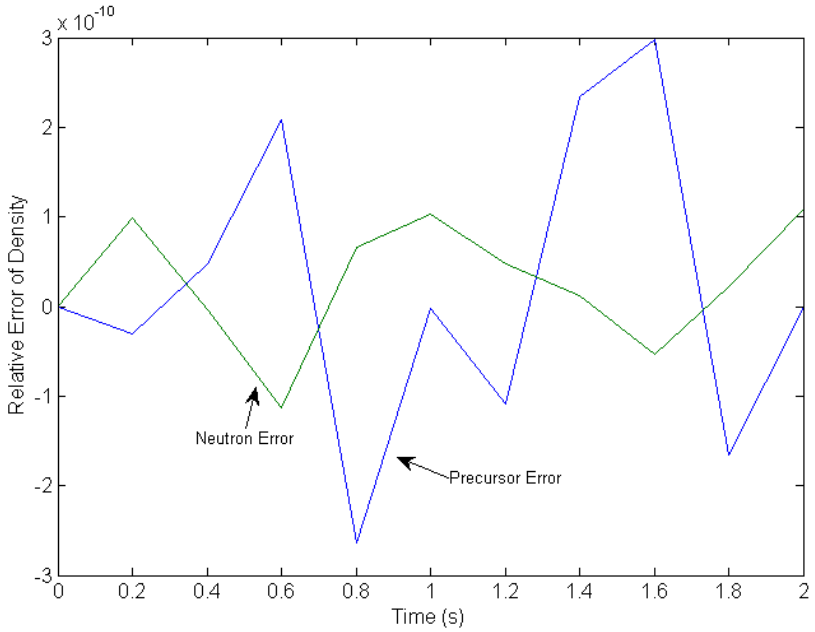

Figure 13: Error Development within Sinusoidal

Reactivity Test 2 Periods 1st Order

By simply increasing the order, the error of the neutron and precursor densities is immediately improved. The plot is similar to Figure 4 in a few ways: there seems to be no pattern, the neutron and precursor densities are of the same magnitude, and there are positive and negative values. However, the magnitude of precision loss for our sinusoidal case is greater by a couple of orders of magnitude. It isn't fair to necessarily state that the linear results produce less error at this point as the step sizes of our two problems are different as well. Instead, an in depth analysis is done to provide insight on step size and convergence rate in VI.F and VI.G.

VI.E. Verification and Error Accumulation: Periods of Sinusoidal Reactivity

Now that some understanding of the local sinusoidal error is established, the properties of many periods are of interest in terms of how the error will accumulate over time. The same conditions from the previous section are used, but calculate out to 50 periods instead of 2. These are the slightly modified code parameters used and the resulting graph.

Table 14: Code Parameters Used for Sinusoidal Reactivity Test 50 Periods 0^{th} Order

Δt	0.01
Number of Steps	5000
$\rho(t)$	$0.001\sin(6.28t)$
$S(t)$	0
Poly. Order	0

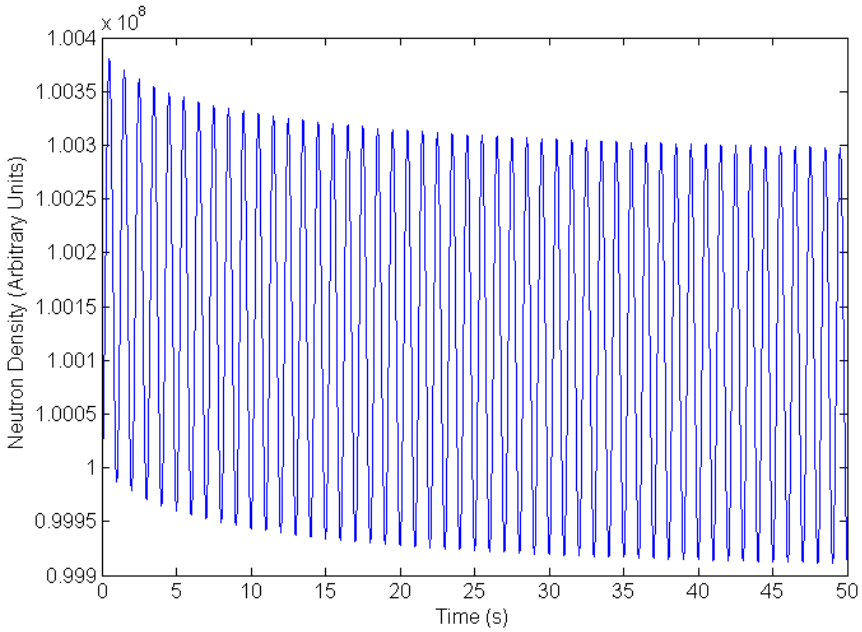

Figure 14: Sinusoidal Reactivity Test

Results 50 Periods 0^{th} Order

With many more periods, the behavior of the result can be characterized. An initial observation is that each period is slightly lower than the previous one, a property first observed in the analysis of 2 periods. However, as time continues, the trend seems to reach some equilibrium sinusoid. The average value during this equilibrium is above the starting steady state value. However, the trough of each wave dips well below the starting value. This overall behavior is created by the interactions of having a sinusoidal reactivity. A regular sine wave has positive reactivity early and negative reactivity later. The precursors lag behind the sine wave and do not contribute as much as required during the negative

portion. The result is a net loss of neutrons. As time passes by, this effect fades as each sine may look like similar to any of its shifted versions. Although not shown here, the same reactivity function was studied after being shifted by increments of $\pi / 2$. This is the equivalent of a cosine wave, negative sine wave and negative cosine wave. The negative sine wave looks similar, but flipped over an imaginary line equal to the starting density. Now that the negative reactivity comes early, the precursors are always larger than required and slowly increase the neutron population. Thus, the overall average is below the starting density, but after each period, the neutron population is above the starting density. For cosine reactivity, there is some positive reactivity early and late, but negative in between. The reverse is true as well. The resulting graphs are also flipped over the starting density, but they do not have any net gain or loss. Instead, a simple sine wave was observed.

Although these are interesting behaviors generated by the nature of the PRKEs, our goals guide our interests towards the accumulated error of this test case and how our polynomial approximation order affects that error. First, the neutron density error is examined.

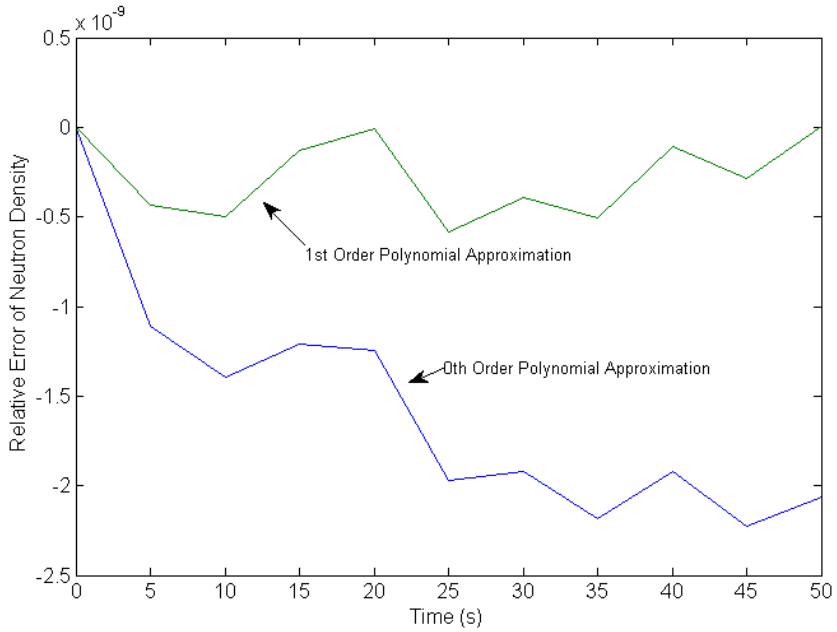

Figure 15: Neutron Error Development within Sinusoidal Reactivity Test 50 Periods

Our sampling rate allows the error to be calculated after every 5 periods. Unlike the increasing linear reactivity cases, the algorithm is now generating negative error, or predicting densities that are lower than the actual value. The 0^{th} order value seems to be generating more negative error over time. This implies that using constants to approximate the sinusoid incrementally contributes error. The 1^{st} order approximation does not have this problem and maintains approximately the same order of error that was seen in the case of 2 periods. If there are any errors that have the potential to accumulate, they most

likely cancel out when the reactivity changes sign. Next, the precursors are examined.

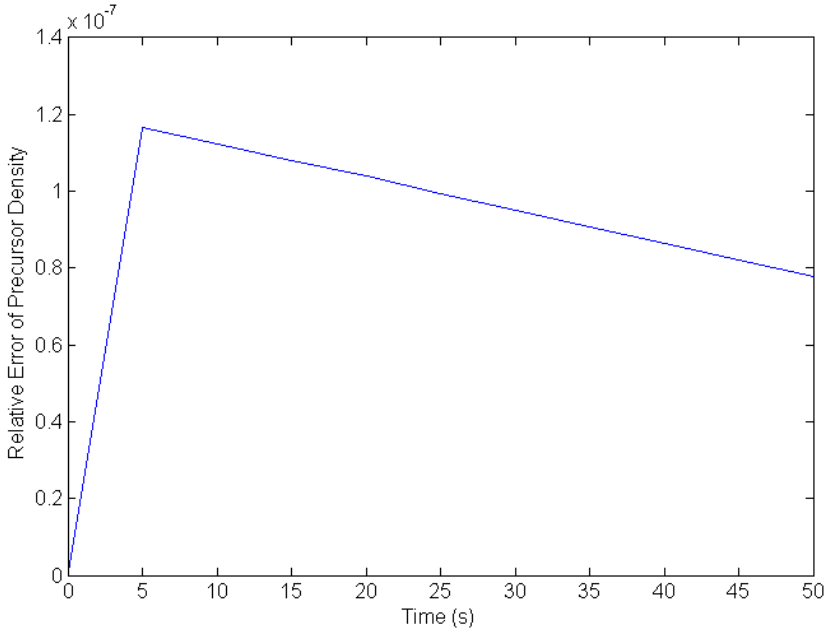

Figure 16: Precursor Error Development within Sinusoidal Reactivity Test 50 Periods 0^{th} Order

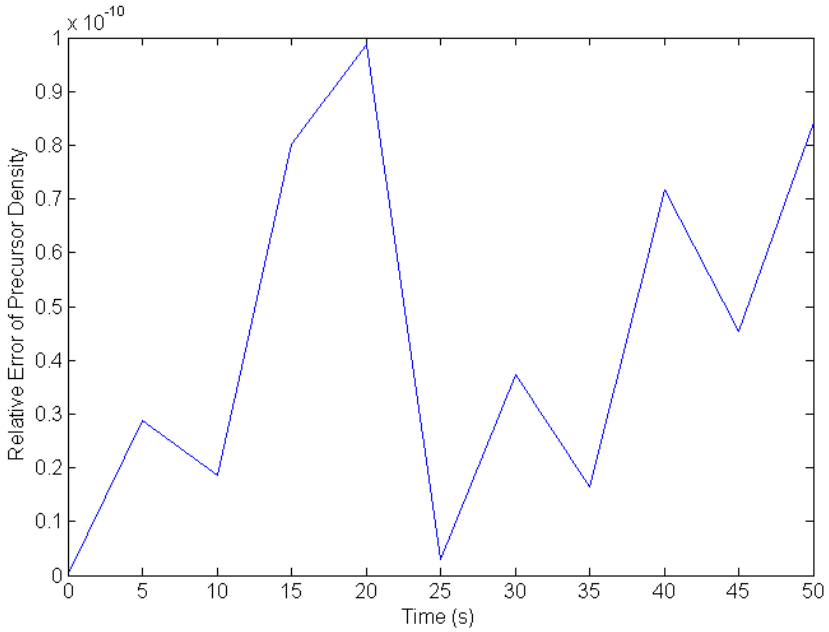

Figure 17: Precursor Error Development within Sinusoidal Reactivity Test 50 Periods 1st Order

The 1st order graph of the precursors is what one would expect. The magnitude of the error is some low amount approximately the same magnitude as the 1st order neutron density error. The error walks randomly, but never becomes negative in our sample. This may be a coincidence due to low sampling rates.

In comparison, the 0th order graph has a much greater magnitude of error. This result is quite interesting. The initial kink is created because there is no error at $t = 0$. However, the error decreases after the first jump almost perfectly in a linear fashion. The sampling system does calculate error on the same point

of the wave at every given point because the objective is to see the behavior of the whole picture (behavior of each individual wave is discussed in VI.D), which allows the possibility of such an observation. Perhaps more information can be gained with some magnification in sampling of the earlier portion of the graph. Additionally, the linear portion of the error is cut off after 50 periods. Some mechanism seems to be removing error generated in the early periods. Eventually the error will probably be dominated by some other source and the linear pattern will deviate. To truly understand the influence of the order of polynomial approximation has over the relative error, it must be observed in conjunction with the step size. This is done in VI.G.

VI.F. Convergence Test: Linear Reactivity

One important feature of our method is its ability to converge to the correct value. The verification tests show that our algorithm can produce values with relative errors almost as low as there are digits of precision. In order to quantify the performance of our method itself, the rate of convergence must be mapped as a function of step size. It is assumed that relative error follows the form given in equation (99). The constant p gives us the order of convergence. Logarithms can be used in order to create a form that makes solving for the constants easier:

$$\log(\varepsilon) = \log(a) + p\log(\Delta t). \tag{114}$$

With a list of step sizes and their associated errors, linear regression can be used in order to determine the constants. The same conditions are used for linear reactivity verification tests over a variety of step sizes. The tested step sizes are listed as

Table 15: Convergence Test Step Sizes for Linear Test Case

Step Count	Δt
2	50
5	20
10	10
50	2
100	1
500	0.2
1000	0.1
5000	0.02
10000	0.01
50000	0.002
100000	0.001

The result of running these different cases is shown in Figure 18 and Figure 19.

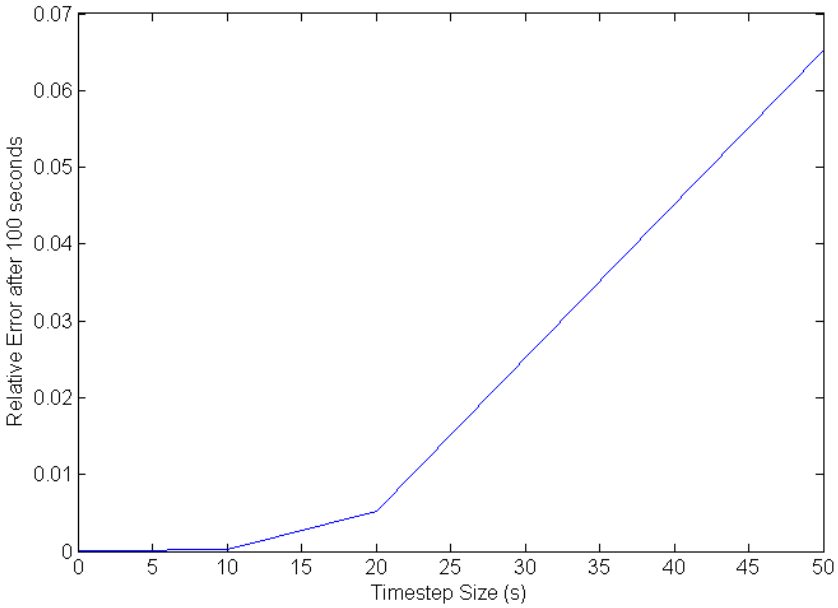

Figure 18: Relative Error of Linear Reactivity Test

Case as a Function of Step Size

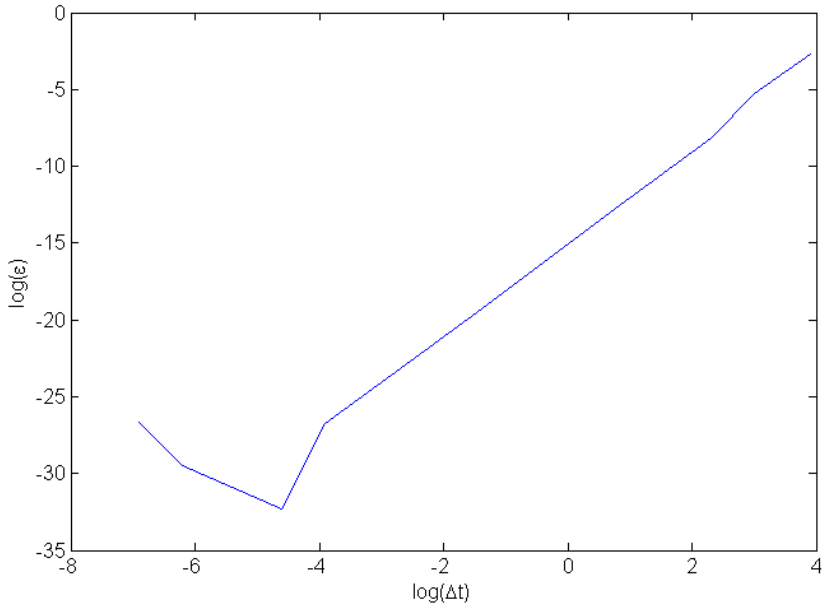

Figure 19: Log of the Relative Error of Linear Reactivity Test

Case as a Function of Log of Step Size

In our log plot, one may notice a decent linear region. However, for lower time steps the error no longer follows a linear pattern. This is explained by the fact that the error cannot fall below the precision level within the code itself. Instead loss of precision error dominates over the error that would accumulate in the algorithm. These points are removed allowing a linear regression to be calculated using the remaining points.

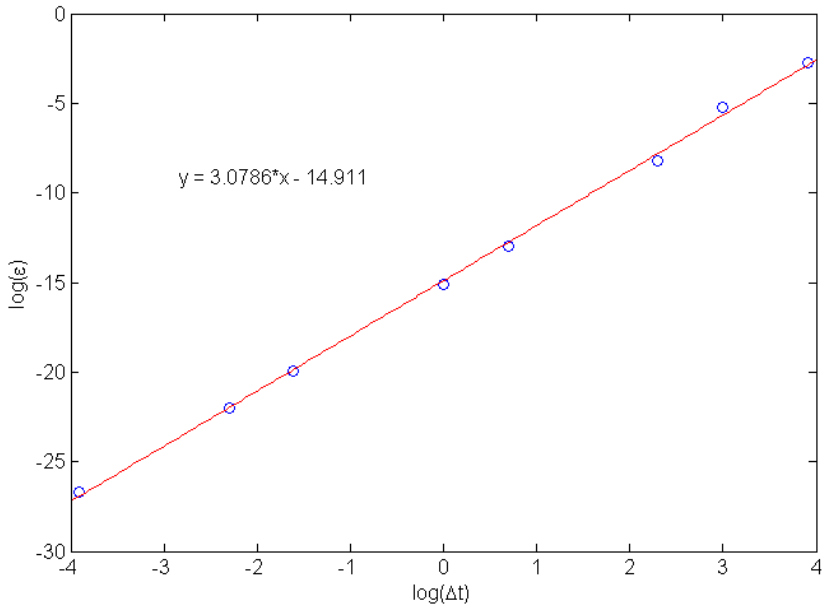

Figure 20: Linear Regression of Linear Reactivity Log Error Plot

The regression produces a strong fit. Using the values solved, one can calculate the constants for the relative error in linear reactivity.

$$\varepsilon \sim 3.3437 \times 10^{-7} (\Delta t)^{3.0786} \tag{115}$$

The method produces slightly better than 3rd order convergence.

VI.G. Convergence Test: Sinusoidal Reactivity

The rate of convergence for sinusoidal reactivity could differ from the linear case because there is an additional source of relevant error. A different set of step sizes are used that are more appropriate for a 50 seconds/50 periods case study. Additionally, 0th order and 1st order polynomial approximations are used separately in two different test cases. The list of step sizes used is given below.

Table 16: Convergence Test Step Sizes for Sinusoidal Test Case

Step Count	Δt
25	2
50	1
100	0.5
500	0.1
1000	0.05
5000	0.01
10000	0.005
50000	0.001

These step sizes are used in the same procedure as before: take the log of error and time step, remove the points that are from alternative sources of error, and create a linear regression of the remaining points.

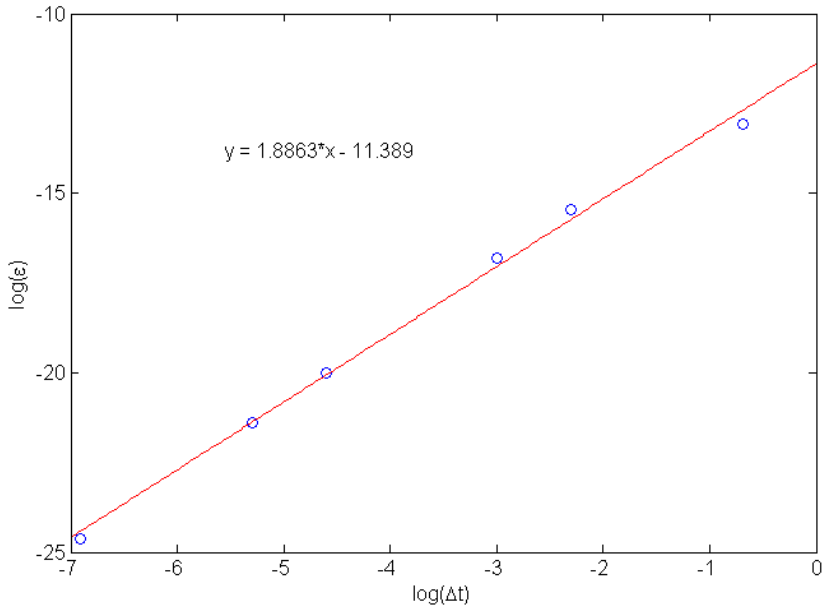

Figure 21: Linear Regression of Sinusoidal Reactivity

0^{th} Order Log Error Plot

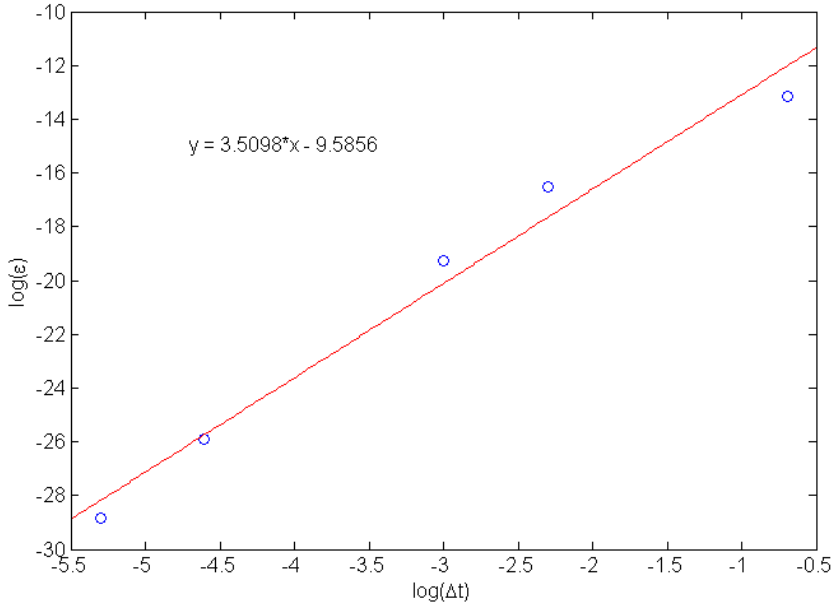

Figure 22: Linear Regression of Sinusoidal Reactivity

1st Order Log Error Plot

These results provide us with the constants for relative error for 0th order

$$\varepsilon \sim 1.1319 \times 10^{-5} (\Delta t)^{1.8863} \tag{116}$$

and 1st order

$$\varepsilon \sim 6.8711 \times 10^{-5} (\Delta t)^{3.5098} . \tag{117}$$

The 1st order polynomial approximation for the sinusoidal reactivity test case produces an excellent convergence rate; about halfway between 3rd and 4th order. However, the 0th order approximation produces a convergence rate shy of 2nd order.

VI.H. Fidelity of Results for Large Time Steps

One of the goals was to observe the result of using large time steps and analyze their use. Rather than create a new test case, the tests in VI.F and VI.G should be sufficient for some discussion on this matter. How useful a result is depends on the requirements of the problem itself. In many of our test cases, large time steps were used. These time steps are relatively large compared to all of the precursor lifetimes and especially large compared to the neutron lifetime. For some especially large time steps, such as the ones taken in VI.F, one may note that only a few digits of precision remain. These calculations may be useful for interpreting the overall behavior of a problem set, especially if it is mostly well behaved. If there are any odd behaviors in the reactivity or source term, it may be wise to individually solve those sections with the appropriate step size. The convergence tests show how the relative error varies with time step size. Those equations can roughly calculate the largest step size allowed if given an error limit. Some numerical methods have a quality that slows the divergence of the error with respect to step size on a log-log plot. That characteristic was not observed with this method. If it exists, it is outside the scope of step sizes tested.

VI.I. Verification: Error Control Scheme

Several attributes were sought after while designing the error control scheme. Overall, the goal of the error control is to reduce the cost of calculation while still maintaining an accuracy goal. The error control scheme should produce the correct result regardless of how crude it is. The linear reactivity test case was used for a time period of 10 seconds with a variety of tolerances and starting Δt values in order to verify the final answer given, and to observe the effect of tolerances on the performance and accuracy of the result. These effects are explained in I.D.6. Note that each step is composed of 3 passes of our algorithm.

Table 17: Validation and Performance Check of Error Control Scheme

Case	Δt_i	$\text{relTol}_{\text{half}}$	$\text{relTol}_{\text{doub}}$	ε	Steps Taken
1	0.01	1×10^{-4}	1×10^{-11}	8.84×10^{-11}	35
2	2	1×10^{-4}	1×10^{-11}	1.42×10^{-07}	3
3	10	1×10^{-4}	1×10^{-11}	-5.64×10^{-7}	1
4	0.01	1×10^{-9}	1×10^{-11}	8.29×10^{-11}	32
5	2	1×10^{-9}	1×10^{-11}	3.29×10^{-10}	20
6	10	1×10^{-9}	1×10^{-11}	8.84×10^{-11}	35
7	0.01	1×10^{-4}	1×10^{-7}	3.12×10^{-5}	21

The relative error produced in each case is lower than the halving tolerance, which was our original goal. One must keep in mind that the halving tolerance keeps the error down for the current iteration, but error accumulates over time. The halving tolerance will eventually be exceeded once a sufficient amount of error has accumulated. A quick discussion of each of these cases will confirm our predictions of how the interaction of the two tolerances will affect performance and accuracy. The first case has a very lenient halving tolerance, but the doubling tolerance gives little room to improve the performance. The result is an

especially accurate result with comparably poor performance time. The second and third cases approach the same tolerances using a large time step instead. Due to the fact that the error is low already, the error control scheme just doubles the 2 second step size once and does nothing to the 10 second time step. Cases 4, 5 and 6 have very strict tolerances. No matter what initial step size used, the algorithm reduces it in order to meet the accuracy goal. The initial time step size of 2 is coincidentally close to some multiple of 2 to the optimal step size for the given accuracy goal. The result is fewer steps are required. This is a good example of motivation towards a more robust error control scheme than the current solution. The final case loosens up the halving tolerance but starts with a low step size. The result is several instances of doubling in order to quickly solve the problem. As a comparison, if the error control scheme is turned off, the same case would take 1000 passes of the algorithm. Each iteration of the error control scheme requires 3 passes, for a total of 63 passes; a massive increase in performance while still meeting accuracy goals.

VI.J. Case Study: Prompt Criticality

It is expected that our error control activity is a function of the time scales present within a problem set. Prompt criticality problems provide many magnitudes of change in population growth rate. This makes them an excellent case for revealing the limitations of our error control scheme. Time spans are

initially dominated by the precursors, but eventually the neutrons can sustain themselves and the rate of increase in population drastically changes. If short time steps are used for the entire process, the computational costs will be high. If longer time steps are used, fidelity is lost during the portions that required a closer look. Additionally, it is difficult to know what magnitude of step size will accomplish a specified accuracy goal. A well functioning error control scheme will attempt to address these issues.

As stated at the beginning of the testing section, a different neutron lifetime is used for this case study, resulting in different initial conditions entirely. A neutron lifetime of $\Lambda = 3 \times 10^{-5}$ s is used. This results in the following densities.

Table 18: Prompt Criticality Initial Conditions

n	100000000.000000
c_1	55555555555.55
c_2	157253599114.0
c_3	37797619047.62
c_4	28460686600.22
c_5	2192982456.14
c_6	299003322.26

These initial conditions represent a critical steady state without source. Reactivity is then linearly increased and eventually it passes the limit for prompt criticality. These initial conditions are represented in our code by the following parameters.

Table 19: Code Parameters Used for Prompt Criticality Test

Δt	0.0001
Number of Steps	150000
$\rho(t)$	$0.0005t$
$S(t)$	0

With these initial conditions, prompt criticality will be achieved in approximately 13 seconds. The first test run untilizes no error control in order to get an idea of the behavior of this system, and to provide additional verification to the original algorithm itself. The neutron density for this system is shown in Figure 23.

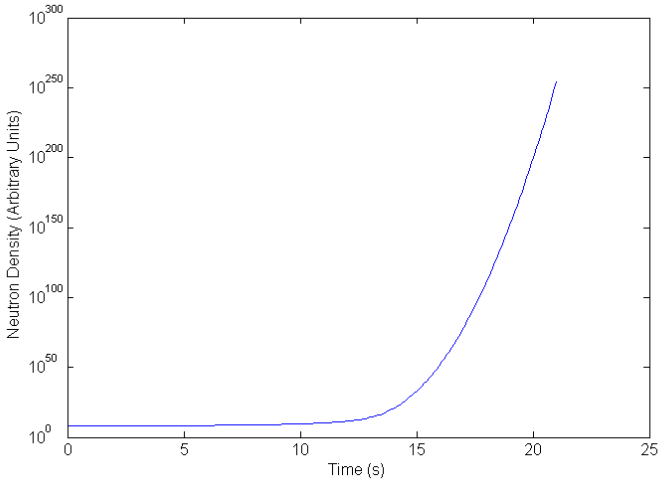

Figure 23: Neutron Density Prompt Criticality Test

Strict time steps are used in order to minimize the accumulation of error. In order to verify this, the error is monitored as time progresses.

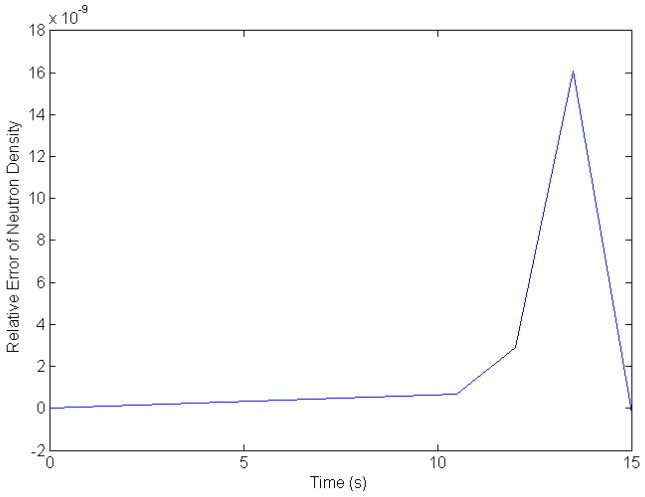

Figure 24: Neutron Density Error Prompt Criticality Test

One may observe that many significant digits are maintained. The approach to the prompt critical region may prove to be problematic. At first, significant positive error is generated. Afterwards, some mechanism is causing negative error to build up. However, these results verify that the algorithm keeps a sufficient number of digits of precision without the error control scheme. Now, the error control scheme can be tested for how much accuracy is sacrificed by each step saved.

The first error control test will use lenient tolerances. The following parameters were used.

Table 20: Error Control Parameters Used for Prompt Criticality Test

Δt_i	0.001
$\text{relTol}_{\text{half}}$	1×10^{-5}
$\text{relTol}_{\text{doub}}$	1×10^{-8}

Using these tolerances and running the same problem conditions, the code used 1,700 iterations of the error control mechanism. This is the equivalent of 5,100 passes of the algorithm. This is compared to the 15,000 passes that would have to be done without error control, or 150,000 passes done in the previous

verification example. One can observe the accumulation of error in the following graph.

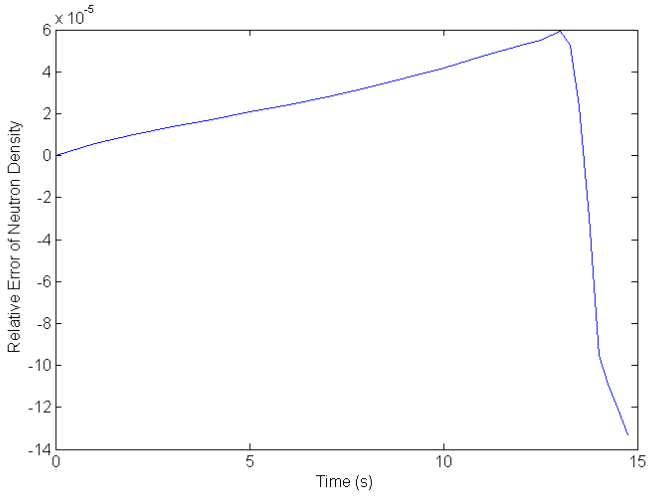

Figure 25: Prompt Criticality Error Control Test 1

As with the case with no error control, positive error is generated before going critical and negative error afterwards at a rapid rate. The total amount of error accumulated at the end of the solution is an order of magnitude larger than our tolerance. However, it was discussed that the halving tolerance just monitors the error within its current iteration. Error is accumulated in a linear fashion, so eventually the total relative error would exceed the halving tolerance given a sufficient number of iterations. In this case, 1,700 iterations resulted in an error higher than the tolerance. Perhaps a more viable accuracy goal would be comparing the tolerance with the relative error divided by the total number of iterations. The total amount of accumulated error only left about 5 digits of

precision left. Although this is enough to get an idea of the general behavior of our problem case, most applications will require better accuracy. After fully analyzing the results from our current tolerances, additional insight can only be gained by analyzing the same problem using strict tolerances.

One may predict that the growth rate of populations would influence the error accumulated in addition to the time step size. Although the interaction is probably too complicated to easily map out, some features can be observed. In order to do so, the step size as a function of time reveals where the error control mechanism took action.

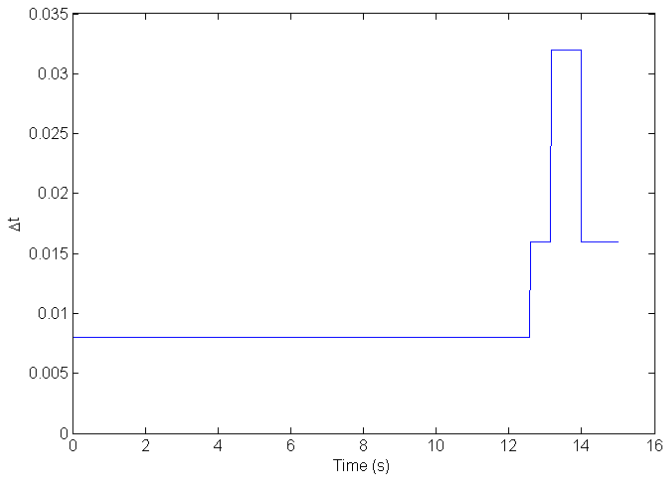

Figure 26: Error Control Step Sizes Prompt Criticality 1

As expected, a shift is observed where prompt criticality becomes more relevant. As the precursors become less of a factor, the solution mimics a much simpler solution involving just the reactivity and neutron lifetime. The step size

can then increase. However, reactivity continues to grow at a linear rate, and neutron population grows at time scales equivalent to its lifetime. Due to the fact that the reactivity isn't a constant, an exponential approximation isn't exact and eventually smaller time steps are required to accurately map out the behavior of the system. The error control scheme will change the step size in order to compensate for this. The relationship between the rate of growth and error control activity may provide some insight on this prediction. The rate is given by

$$\frac{d}{dt}\log\Big(P(t)\Big) = \frac{\dot{P}}{P}. \tag{118}$$

The code generates a list of all populations of interest with their associated timestamps as the algorithm runs. This information can be used to create an approximation using a backwards finite difference.

$$\frac{\dot{P}}{P} \approx \frac{n_k - n_{k-1}}{n_k \Delta t} \tag{119}$$

This approximation is applied across the compilation of data produced by the code.

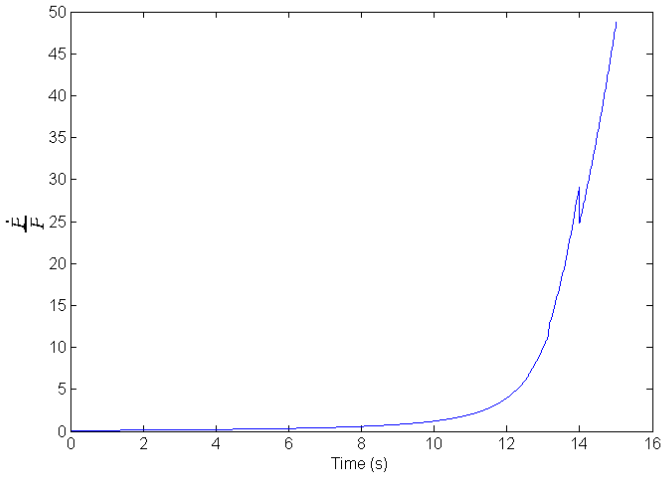

Figure 27: Rate of Growth Prompt Criticality 1

The minor hiccup in the graph was created by the error control scheme shifting to another time step. This interaction caused the finite difference poorly approximate the derivative. However, the goal was to examine the shape of the graph and compare it to the activity of the error control scheme. The error control scheme seems to use small time steps early where precursors determine the growth rate. Additionally, the error control scheme favors small time steps later when the growth rate is large and increasing.

Finally, this analysis is repeated with strict tolerances in order to try and mitigate some of the error found in this case study. Additionally, the time of interest is increased to 21 seconds in order to further analyze what happens after prompt criticality. The following tolerances in Table 21 are used.

Table 21: Error Control Parameters Used for Prompt Criticality Test

Δt_i	0.001
$\text{relTol}_{\text{half}}$	1×10^{-9}
$\text{relTol}_{\text{doub}}$	1×10^{-10}

Using those parameters resulted in 9,512 iterations of error control, or almost 30,000 passes of the algorithm. This is compared to the 21,000 passes that would have occurred if error control was not used. Unfortunately, this is a rare example (associated with long runs) where error control ends up hindering the performance more than it helps. However, error control still is useful here for showing approximately what order of magnitude of step size is necessary in order to achieve certain accuracy goals. As before, the accumulation of error is of interest. The first 15 seconds are compared with the previous test case.

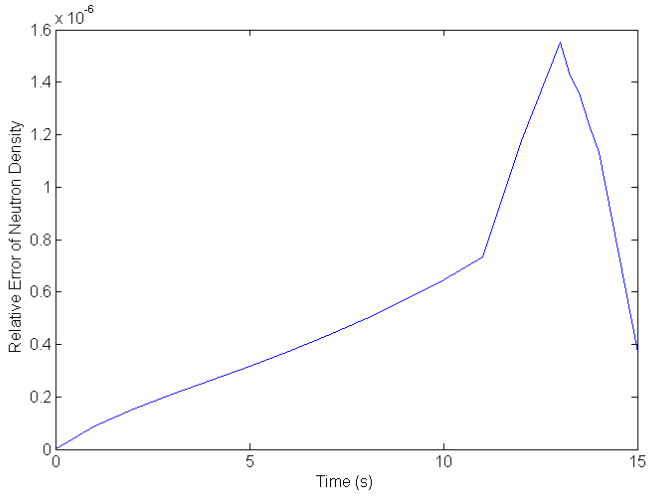

Figure 28: Prompt Criticality Error Control Test 2

The overall error is better than the previous test, but only by a couple of orders of magnitude. The tolerance is once again larger than the total accumulated error, but less than the relative error divided by the number of iterations. Next the step size as a function of time is analyzed to observe where the error control tolerances are met.

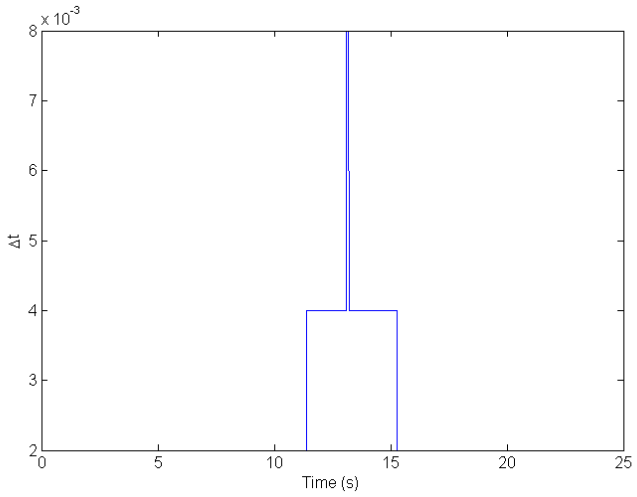

Figure 29: Error Control Step Sizes Prompt Criticality 2

The result is similar to the previous case, with overall smaller step sizes. Additionally, another reduction of step size occurs after the 15 second mark. This supports our claim that shorter step sizes must be used well after prompt criticality to accurately map the behavior of the problem. Finally, the growth rate of the neutrons is observed in Figure 30.

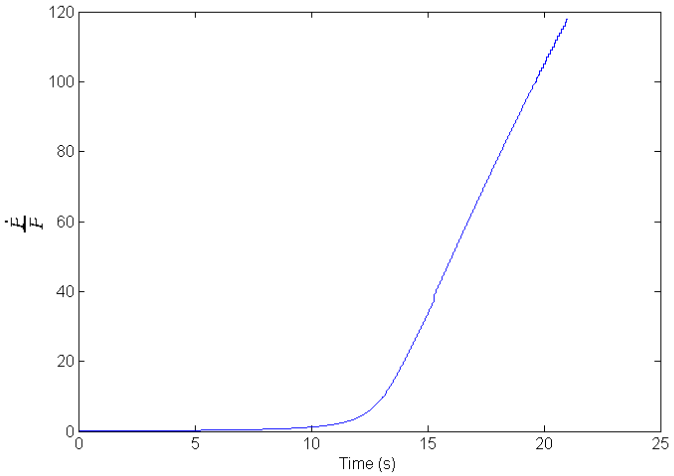

Figure 30: Rate of Growth Prompt Criticality 1

With smaller step sizes, there are no longer any blatant discontinuities from our finite difference. Comparing this figure with the time stamps of step size changes, the observations of our error control scheme are confirmed; more steps are needed for precursor dominated regions and regions with high changes in growth rate.

VII. CONCLUSION

A method using exponential moment methods in order to solve the PRKEs was designed. The goals were oriented towards two major categories, accuracy and performance. Exponential moment methods provide opportunity for improvements in accuracy and performance, providing our main motivation for exploring their applications. The features presented within our method were designed to progress towards one or both of these categories. The algorithm was coded in FORTRAN in order to provide a platform for testing. Test cases were designed to evaluate both the algorithm as a whole and individual design features provided by the algorithm.

For our accuracy oriented goals, a series of verification tests were designed. Our results show that our algorithm converges to the correct value for the cases tested and discussed. The designed error control scheme was also tested for verification purposes. Solutions provided by the error control scheme produced relative errors less than the tolerance per iteration in our test cases.

Our performance based goals, convergence rate and cost analysis tests were used. The rate of convergence of the base algorithm depends on the test case varying from third to fourth order. Our sinusoidal suite can provide performance of similar caliber with well selected polynomial approximations. Poor polynomial approximations still converged to the correct value, but at a much slower rate.

In order to improve performance, the error control scheme was designed to minimize computational costs given an accuracy goal. The error control scheme was successful in most cases; however, it was noted that the accuracy goal was only effective for individual iterations. For longer runs, accumulation of error can jeopardize the accuracy goal. The performance of the error control scheme is a function of selected tolerances and starting conditions. For many cases, the error control scheme was effective in reducing the total number of calculations taken. However, the overall design of the error control scheme is crude and inflexible, and many aspects can benefit from further improvement.

VII.A. Future Work

The process of design and testing of our method revealed several opportunities for improvement of our algorithm.

- Our approximation for the neutron density is exponential in form with two degrees of freedom. The approximation was sufficient in converging to the correct value with appropriate step sizes and fulfilled the goal of producing solutions that are strictly positive. One feature of our exponential approximation is positive concavity. Due to the fact that the outer constant, a, is always positive, the second derivative of our approximation will also always be positive. Moment matching is then used to fit the exponential approximation. For situations where the neutron population has negative concavity, perhaps this leads to preventable accumulation of error. Although moment matching is

effective at mitigating much of this error, perhaps a third degree of freedom would provide additional flexibility. However, the repercussions on the derivation of our algorithms have not been considered. One repercussion that can easily be imagined is that a form that encorporates negative concavity will allow negative solutions without some strict conditions. The proposed form for testing could perhaps be

$$\tilde{n}(t) = a + be^{\alpha t} . \tag{120}$$

- The effects of accumulation of error and the relationship to reactivity have been somewhat observed throughout this work. In general, it was observed that moments that favored more reactivity early and less at later times would accumulate negative error. The reverse is true as well. Additionally, in cases where reactivity varies from positive to negative resulted in the normal sources of accumulating error to mitigate each other. These observations may provide a starting point for some research of a higher ordered method.
- The properties of the error control scheme reveal crude design features that can be improved. The method of estimating error is somewhat arbitrary and can possibly be fooled by certain test cases where the estimation is relatively low compared to the actual error. This may be the case in prompt criticality, but further investigation is required to warrant this conclusion. Perhaps a higher ordered method of error estimation would improve the ability of our error control scheme to make decisions correctly against the provided tolerances. One method is taking a Picard iteration using the exponential approximation once

the constants are known. Additionally, the error control scheme has a poor adaptive structure. Any changes in step size are done by factors of 2. The resulting step size can be far from optimal. With a robust error estimation scheme, an algorithm can be created that selects optimal step sizes. This improved algorithm would then just need a single tolerance in the form of an accuracy goal. Such an improvement is well beyond the scope of this research.

- The time step used for our approximation is also the time step used for approximating sinusoidal reactivity using polynomials. Higher ordered polynomial approximations are useful for larger portions of the sinusoid where curvature becomes relevant. Before the time step becomes large enough to benefit from this, the error is dominated by the original algorithm. By separating these two factors, the sine can be approximated for a certain time step and many smaller time steps of that fitted polynomial can define the solution.

VIII. APPENDICES

VIII.A. Picard Iteration for Improved Error Control

After each pass of our algorithm, there is now have an estimate for the neutron population within our time step. The estimation is given by the knowledge of a and α within equation (5). Our normal solution is to substitute these values into equation (23). This method solves an approximation from the actual problem. Knowing that (23) is a higher ordered approximation for neutron population than (5), one can solve the first Picard iteration, eliminating some of the higher ordered errors. This method would be useful in an error control scheme. In order to do so, equation (23) is substituted into (14) after solving for a and α. Using some exponential moment methods, one can reduce the equation to arithmetic. Then the precursors can be compared to the original algorithm and relative errors can be calculated. A further extension of this would be substituting the precursors back in and solving for neutrons again. This process can be repeated for multiple Picard iterations. Unfortunately, the testing and feasibility of using Picard iterations for the purpose of error control was beyond the scope of this project. In order to give some insight to this technique, the solution will be given of calculating the precursors to a higher order by substituting (23) into (14). The solution is untested, but may give a decent starting point for future works.

$$
\begin{aligned}
c_i\left(t'\right) = c_i\left(0\right)e^{-\lambda_i t'} &+ \frac{\beta_i}{\Lambda}\int_0^{t'} e^{-\lambda_i\left(t'-t''\right)} n\left(0\right)e^{-\overline{\kappa}t''}dt'' \\
&+ \frac{\beta_i}{\Lambda}\int_0^{t'} e^{-\lambda_i\left(t'-t''\right)}\sum_{ii}\lambda_{ii}c_{ii}\left(0\right)\left[\int_0^{t''} e^{-\overline{\kappa}\left(t''-t'''\right)}e^{-\lambda_{ii}t'''}dt'''\right]dt'' \\
&+ \frac{\beta_i}{\Lambda}\int_0^{t'} e^{-\lambda_i\left(t'-t''\right)}\left[\int_0^{t''} e^{-\overline{\kappa}\left(t''-t'''\right)}S\left(t'''\right)dt'''\right]dt'' \\
&+ \frac{\beta_i}{\Lambda}\int_0^{t'} e^{-\lambda_i\left(t'-t''\right)}\left[ae^{-\overline{\kappa}t''}\int_0^{t''} e^{\left(\overline{\kappa}+\alpha\right)t'''}\delta\kappa\left(t'''\right)dt'''\right]dt'' \\
&+ \frac{\beta_i}{\Lambda}\int_0^{t'} e^{-\lambda_i\left(t'-t''\right)}\left[a\sum_i\frac{\lambda_i\beta_i}{\Lambda}\int_0^{t''} e^{-\overline{\kappa}\left(t''-t'''\right)}\left[\int_0^{t'''} e^{-\lambda_i\left(t'''-t''''\right)}e^{\alpha t''''}dt''''\right]dt'''\right]dt''
\end{aligned}
\tag{121}
$$

The precursors are divided into 5 major integrals that need to be converted into moment functions. Rather than showing the lengthy process the untested solutions of each integral will be given.

$$
c_{i1}\left(t'\right) = \frac{\beta_i n(0)t'}{\Lambda}e^{-\lambda_i t'}\mathcal{M}_0(\overline{\kappa}t' - \lambda_i t') \tag{122}
$$

$$
c_{i2}\left(t'\right) = \frac{\beta_i (t')^2}{\Lambda}\sum_{ii}\lambda_{ii}c_{ii}(0)e^{-\lambda_i t'}\mathcal{M}_0\left((\overline{\kappa}-\lambda_i)t', (\lambda_{ii}-\lambda_i)t'\right) \tag{123}
$$

$$
c_{i3}\left(t'\right) = \frac{\beta_i (t')^2}{\Lambda}\sum_{p=0}^{P_s} s_p (t')^p \mathcal{M}_p(\overline{\kappa}t', \lambda_i t') \tag{124}
$$

$$
c_{i4}\left(t'\right) = \frac{a\beta_i (t')^2}{\Lambda}\sum_{p=0}^{P_\rho} \partial k_p (t')^p e^{\alpha t'}\mathcal{M}_p\left((\overline{\kappa}+\alpha)t', (\alpha+\lambda_i)t'\right) \tag{125}
$$

$$
c_{i5}\left(t'\right) = \frac{a(\beta_i)^2\lambda_i}{\Lambda^2}(t')^3 e^{\alpha t}\mathcal{M}_0\left((\lambda_i+\alpha)t', (\alpha+\overline{\kappa})t', (\alpha+\lambda_i)t'\right) \tag{126}
$$

Thus, one can have a higher order estimate of the precursor population using

$$
c_i\left(t'\right) = c_i\left(0\right)e^{-\lambda_i t'} + c_{i1}\left(t'\right) + c_{i2}\left(t'\right) + c_{i3}\left(t'\right) + c_{i4}\left(t'\right) + c_{i5}\left(t'\right) \tag{127}
$$

VIII.B. FORTRAN Code

A few notes about the FORTRAN code:

- The FORTRAN code was originally designed to just run the algorithm. The additional testing required for add on modules littered some of the overhead structure.
- Some variables are poorly named (mainly del_T, this is delta T or Δt). Some temporary variables are used in order to ease some computation. This may create some difficulty when following the code.
- Some methods that were either rejected or improved upon still remain in the code in the form of comments. These were used for reference and testing purposes.

VIII.B.1 Main

```
Program RPK
!*****************************************************************************
!Program: main execution
!Purpose:   Calculate n(del_T) and c_i(del_T) given inital n and C_i,
!           and source/reactivity info
!Created: 16 Nov 2011
!Version: 1.2
!*****************************************************************************

Use User_Data, Only:    Get_Working_Directory, &
                        & Get_Problem_Data, & !N(0), C_i(0), S(0), rho(0), del_T, betai, Ci
                        & Source_and_Reactivity_Calc
Use SolutionSteps, Only:  SolveB, SolveAlpharobust, SolveA, SolveNC
```

```
Use Finalize, Only: RecordSolution, Initial_Condition_Reset
Use Variables, Only: Test, stepcount, Directory, OutputFolder, &
                    & currentstep, N0, C_i0, Ndt, C_idt, del_T, &
                    & EC, relTolhalf, relToldoub, Tfinal, Ntest, &
                    & currenttime, C_i0O, S_CoeffO, rho_CoeffO, &
                    & N0O, S_Coeff, rho_Coeff, del_Tmin
Use Kinds, Only: dp

Implicit None

    Character(120)::OutputFile
    Integer :: i

    !For sampling in EC
    Real :: SampleT
    Logical :: ECsampleSwitch =.True.
    Integer:: ECsamplei


!Setting up problem

    Call Get_Working_Directory
    Call Get_Problem_Data

!Done setting up problem

!Creating the output files for recording solution

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'n.txt'
    Open(unit=27, file=OutputFile)

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'c1.txt'
    Open(unit=21, file=OutputFile)
```

```
    OutputFile = Trim(Directory)//Trim(OutputFolder)//'c2.txt'
    Open(unit=22, file=OutputFile)

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'c3.txt'
    Open(unit=23, file=OutputFile)

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'c4.txt'
    Open(unit=24, file=OutputFile)

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'c5.txt'
    Open(unit=25, file=OutputFile)

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'c6.txt'
    Open(unit=26, file=OutputFile)

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'time.txt'
    Open(unit=28, file=OutputFile)

    !Specialty Output
    OutputFile = Trim(Directory)//Trim(OutputFolder)//'n10.txt'
    Open(unit=29, file=OutputFile)

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'c610.txt'
    Open(unit=30, file=OutputFile)

    OutputFile = Trim(Directory)//Trim(OutputFolder)//'time10.txt'
    Open(unit=31, file=OutputFile)

!Done creating output files

!Write initial conditions in output
Write(28,*)"0"
Write(27,*)N0
Write(21,*)C_i0(1)
```

```
Write(22,*)C_i0(2)
Write(23,*)C_i0(3)
Write(24,*)C_i0(4)
Write(25,*)C_i0(5)
Write(26,*)C_i0(6)

Write(29,*)N0
Write(30,*)C_i0(6)
Write(31,*)"0"
!Done writing initial conditions in output

if(EC)then
currentstep=1
currenttime=0._dp
if(ECsampleSwitch)SampleT=Tfinal/200._dp
if(ECsampleSwitch)ECsamplei=1
Do

    N0O=N0
    C_i0O=C_i0
    rho_CoeffO=rho_Coeff
    S_CoeffO=S_Coeff


    !Solve for test value
    del_T=del_T*2._dp

    Call Source_and_Reactivity_Calc
    Call SolveB
        if(Test)write(*,*)"solved B"
    Call SolveAlpharobust
        if(Test)write(*,*)"solved alpha"
    Call SolveA
        if(Test)write(*,*)"solved A"
```

```
    Call SolveNC
        if(Test)write(*,*)"solved NC"

    Ntest=Ndt !Ntest is the full value
    !Done solving test value

    !Solve first of 2 steps
    del_T=del_T/2._dp !Original del_T
    !currenttime=currenttime+del_T

    Call Source_and_Reactivity_Calc
    Call SolveB
        if(Test)write(*,*)"solved B"
    Call SolveAlpharobust
        if(Test)write(*,*)"solved alpha"
    Call SolveA
        if(Test)write(*,*)"solved A"
    Call SolveNC
        if(Test)write(*,*)"solved NC"

    !if(currenttime>=Tfinal)Call RecordSolution
!
!    Write(28,*)currenttime
!    Write(27,*)Ndt
!    Write(21,*)C_idt(1)
!    Write(22,*)C_idt(2)
!    Write(23,*)C_idt(3)
!    Write(24,*)C_idt(4)
!    Write(25,*)C_idt(5)
!    Write(26,*)C_idt(6)

    !if(currenttime>=Tfinal)Exit

    Call Initial_Condition_Reset
```

```
    !Done solving first of 2 steps

    !Solve second step

    !currenttime=currenttime+del_T

    Call Source_and_Reactivity_Calc
    Call SolveB
        if(Test)write(*,*)"solved B"
    Call SolveAlpharobust
        if(Test)write(*,*)"solved alpha"
    Call SolveA
        if(Test)write(*,*)"solved A"
    Call SolveNC
        if(Test)write(*,*)"solved NC"

!    if(currenttime>=Tfinal)Call RecordSolution
!
!    Write(28,*)currenttime
!    Write(27,*)Ndt
!    Write(21,*)C_idt(1)
!    Write(22,*)C_idt(2)
!    Write(23,*)C_idt(3)
!    Write(24,*)C_idt(4)
!    Write(25,*)C_idt(5)
!    Write(26,*)C_idt(6)
!
!    if(currenttime>=Tfinal)Exit

    Call Initial_Condition_Reset

    !Done Solving second step
```

```
    !Error control
    if((Abs(Ntest-Ndt)/Ndt)>relToldoub .AND. (Abs(Ntest-Ndt)/Ndt)<relTolhalf)then  !del_T is just
right
        currenttime=currenttime+(2._dp*del_T)
        if(currenttime>=Tfinal)Call RecordSolution

        Write(28,*)currenttime
        Write(27,*)Ndt
        Write(21,*)C_idt(1)
        Write(22,*)C_idt(2)
        Write(23,*)C_idt(3)
        Write(24,*)C_idt(4)
        Write(25,*)C_idt(5)
        Write(26,*)C_idt(6)

        if(ECsampleSwitch)then
            if(currenttime>(SampleT*ECsamplei))Write(29,*)Ndt
            if(currenttime>(SampleT*ECsamplei))Write(30,*)C_idt(6)
            if(currenttime>(SampleT*ECsamplei))Write(31,*)currenttime
            if(currenttime>(SampleT*ECsamplei))ECsamplei=ECsamplei+1
        end if

        if(currenttime>=Tfinal)Exit
    end if

    if((Abs(Ntest-Ndt)/Ndt)<=relToldoub)then  !del_T is too small, but good answer
        currenttime=currenttime+(2._dp*del_T)
        if(currenttime>=Tfinal)Call RecordSolution

        Write(28,*)currenttime
        Write(27,*)Ndt
        Write(21,*)C_idt(1)
        Write(22,*)C_idt(2)
        Write(23,*)C_idt(3)
```

```
        Write(24,*)C_idt(4)
        Write(25,*)C_idt(5)
        Write(26,*)C_idt(6)

        if(ECsampleSwitch)then
            if(currenttime>(SampleT*ECsamplei))Write(29,*)Ndt
            if(currenttime>(SampleT*ECsamplei))Write(30,*)C_idt(6)
            if(currenttime>(SampleT*ECsamplei))Write(31,*)currenttime
            if(currenttime>(SampleT*ECsamplei))ECsamplei=ECsamplei+1
        end if

        if(currenttime>=Tfinal)Exit
        del_T=del_T*2._dp
    end if

    if((Abs(Ntest-Ndt)/Ndt)>=relTolhalf)then !del_T is too large, bad answer. redo
        del_T=del_T*.5_dp
        !reset conditions
        N0=N0O
        C_i0=C_i0O
        rho_Coeff=rho_CoeffO
        S_Coeff=S_CoeffO
    end if

    if(del_T<del_Tmin)STOP "del_T min reached"

!    if((Abs(Ndt-Ntest)/Ntest)<relToldoub)del_T=del_T*2._dp
!    if((Abs(Ndt-Ntest)/Ntest)>relTolhalf)then
!    currenttime=currenttime-(2._dp*del_T)
!    del_T=del_T/2._dp
!    N0=N0O
!    C_i0=C_i0O
!    rho_Coeff=rho_CoeffO
!    S_Coeff=S_CoeffO
```

```
!    end if
!    if(del_T<del_Tmin)STOP "del_T min reached"
!    !Done Error controlling

    currentstep=currentstep+1
End do

else !Error control off, regular operation

!Looping through steps

Do i=1,stepcount
    !new time step defined here
    currentstep=i

    Call Source_and_Reactivity_Calc
    Call SolveB
        if(Test)write(*,*)"solved B"
    Call SolveAlpharobust
        if(Test)write(*,*)"solved alpha"
    Call SolveA
        if(Test)write(*,*)"solved A"
    Call SolveNC
        if(Test)write(*,*)"solved NC"
    if(i .eq. stepcount)Call RecordSolution

    Write(28,*)currentstep*del_T
    Write(27,*)Ndt
    Write(21,*)C_idt(1)
    Write(22,*)C_idt(2)
    Write(23,*)C_idt(3)
    Write(24,*)C_idt(4)
    Write(25,*)C_idt(5)
    Write(26,*)C_idt(6)
```

```
    if(MOD(currentstep*10,stepcount)==0)Write(29,*)Ndt
    if(MOD(currentstep*10,stepcount)==0)Write(30,*)C_idt(6)
    if(MOD(currentstep*10,stepcount)==0)Write(31,*)currentstep*del_T

    !need previous time step length to go in here for correct condition reset

    Call Initial_Condition_Reset

End Do

End if
!Done looping


End Program RPK
```

VIII.B.2 Variables

```
Module Variables
    !Global Variables
    Use Kinds, Only: dp
    Implicit None

    !************************************
    !create logicals for internal tests or break up modules for testing individual sections
    !or both?
    !************************************
    Logical             :: Test = .False.
    Logical             :: Sinrho
    Logical             :: EC           !Error Control
```

```
!Global Characters
Character(len=120)  :: directory    !directory of the program
Character(len=60)   :: OutputFolder='\Output\'
Character(len=60)   :: InputFolder='\Input\'
Character(len=60)   :: GroupData='GroupInfo.txt'
Character(len=60)   :: ProbData='ProbData.txt' !inital problem data (neutrons, time, etc)
Character(len=60)   :: PrecursorData='PrecursorData.txt'
Character(len=60)   :: SourceData='SourceData.txt'
Character(len=60)   :: RhoData='RhoData.txt'
Character(len=60)   :: ErrorControl='ErrorControl.txt'

!Global Integers
Integer :: n_simpsons               !Interval Number for Simpson's Rule (must be even)
Integer :: b_integral               !subsections for double B0 and B1 integrals
Integer :: TotalGroups              !Number of delayed neutron groups
Integer :: P_S                      !Order of Source Polynomial (and Sp)
Integer :: P_rho                    !Order of Reactivity Polynomial (and dkp)
Integer :: stepcount                !Number of timesteps
Integer :: currentstep


!Allocatable global reals
 Real(dp), Allocatable :: beta_i(:)  !Neutron group birth fraction
 Real(dp), Allocatable :: lambda_i(:)!Neutron Group decay constant
 Real(dp), Allocatable :: C_i0(:)    !Inital Precursor amount
 Real(dp), Allocatable :: C_idt(:)   !Final Precursor amount
 Real(dp), Allocatable :: S_coeff(:) !Coefficients for source term
 Real(dp), Allocatable :: rho_coeff(:)!Coefficients for reactivity
 Real(dp), Allocatable :: Sp_coeff(:)!Coefficients for source/dt
 Real(dp), Allocatable :: dkp_coeff(:)!Coefficients for d_kappa/dt
 Real(dp), Allocatable :: SourceMom(:)!Moments used in source term
 Real(dp), Allocatable :: rho_sin(:) !rho constants for sin function
```

```
    !Global Reals
    Real(dp) :: N0                      !Initial Neutron Concentration
    Real(dp) :: Ndt                     !Final Neutron Concentration
    Real(dp) :: Ndtapprox               !Approximate Final Neutron Concentration
    Real(dp) :: del_T                   !Timestep of interest
    Real(dp) :: N_Lifetime              !Reproductive lifetime of neutron
    Real(dp) :: beta_Tot                !Sum of beta_i (delayed neutron frac)
    Real(dp) :: rho_bar                 !average reactivity in time scale of interest
    Real(dp) :: kappa_bar
    Real(dp) :: B0                      !B0 used for solving a and alpha
    Real(dp) :: B1                      !B1 used for solving a and alpha
    Real(dp) :: a                       !outer coefficient for neutron approximation
    Real(dp) :: alpha                   !exponential coefficient for neutron approx
    Real(dp) :: kbdt                    !kappa bar times delta t

    !Global Reals used in error control

    Real(dp) :: Tfinal                  !Final Time
    Real(dp) :: del_Tmin                !Minimum size for del_T
    Real(dp) :: relTolhalf              !Threshold for changing step size cut in half
    Real(dp) :: relToldoub              !Threshold for changing step size double
    Real(dp) :: Ntest                   !Neutron Concentration test variable
    Real(dp) :: currenttime             !Current time
    Real(dp) :: N0O                     !Original N

    Real(dp), Allocatable :: C_i0O(:)       !Original C_i0
    Real(dp), Allocatable :: S_coeffO(:)    !Original Coefficients for source term
    Real(dp), Allocatable :: rho_coeffO(:)  !Original Coefficients for reactivity


End Module Variables
```

```
Module User_Data
!Handles program set up and variable input
    Implicit None
    Private
    Public Get_Working_Directory, & !Done
           & Get_Problem_Data, & !N(0), C_i(0), S(0), rho(0), del_T, betai, Ci
           Source_and_Reactivity_Calc

    Contains

Subroutine Get_Working_Directory
    Use Variables, Only: Directory
    Implicit None
    Integer:: errortemp
    Character(len=1) check


    Open(Unit=20, File='Directory.txt', IOSTAT=errortemp)
    If (errortemp .ne. 0) Then
        !File does not exist, get directory
       Write(*,*) 'Please specify directory.'
         Read(*,*) directory
     Else
         !Check to see if directory is valid
         Read(20,*,IOSTAT=errortemp) directory
         If(errortemp==0) Then
             Write(*,*) 'Last directory used was:',directory
          Do
```

```
                Write(*,*) 'Is this directory still valid? (Y,N)'
                Read(*,*) check
                Select Case (check)
                     Case ('Y','y')
                          EXIT
                     Case ('N','n')
                          Write(*,*) 'Please specify directory.'
                          Read(*,*) directory
                          REWIND(20)
                          Write(20,*) directory
                          EXIT
                     Case Default
                          Write (*,*) "Please choose 'Y' or 'N' only."
                          CYCLE
                End Select
           End Do
          Else
           Write(*,*) 'Please specify directory.'
           Read(*,*) directory
           Write(20,*) directory
          End If
    End If
    Close(20)
End Subroutine Get_Working_Directory


Subroutine Get_Problem_Data
    Use Kinds, Only: dp
    Use Variables, Only:    InputFolder, Directory, ProbData, &
                            & TotalGroups, C_i0, PrecursorData, &
                            & N0, del_T, N_Lifetime, SourceData, &
                            & RhoData, P_S, P_rho, S_Coeff, &
                            & rho_Coeff, GroupData, beta_i, lambda_i, &
                            & beta_Tot, rho_bar, n_simpsons, &
```

```
                              & kappa_bar, dkp_coeff, Sp_coeff, C_idt, &
                              & SourceMom, b_integral,  &
                              & stepcount, currentstep, rho_sin, sinrho, &
                              & ErrorControl, EC, Tfinal, relTolhalf, relToldoub, &
                              & C_i0O, S_CoeffO, rho_CoeffO, del_Tmin

    !Use Helper, Only: SimpsonsInt
    Use RPKFunctions, Only: rhoFunc, SolveRhoBar

    Implicit none

    Character(120)::WorkingFile
    Integer::i

    !Gather problem Data
    WorkingFile = Trim(Directory)//Trim(InputFolder)//ProbData

    Open(20,file=WorkingFile)
    Read(20,*)N0            !Initial Neutron Conc
    Read(20,*)del_T         !Timestep size
    Read(20,*)stepcount     !Number of timesteps
!    Read(20,*)N_Lifetime    !Neutron Reproductive Life Time
!    Read(20,*)n_simpsons
!    Read(20,*)b_integral
    Close(20)

    !Gather Group Data
    WorkingFile = Trim(Directory)//Trim(InputFolder)//GroupData

    Open(20, file=WorkingFile)
    Read(20,*)TotalGroups
    Allocate(beta_i(TotalGroups))
    Allocate(lambda_i(TotalGroups))
```

```
Do i=1,TotalGroups
    Read(20,*)beta_i(i),lambda_i(i)
End Do

beta_Tot=sum(beta_i)

!Calculate NLifetime instead of using a preset amount (optional)
N_Lifetime = 12.7_dp*beta_Tot + 3._dp*(1._dp - beta_Tot)*(10._dp)**-5;
N_Lifetime = 3._dp*(10._dp)**-5;

Close(20)

!Gather Precursor inital data
WorkingFile = Trim(Directory)//Trim(InputFolder)//PrecursorData

Open(20,file=WorkingFile)
Allocate(C_i0(TotalGroups))
Allocate(C_i0O(TotalGroups))
Allocate(C_idt(TotalGroups))

Do i=1,TotalGroups
    Read(20,*)C_i0(i)
End Do

Close(20)

!Gather Source Data
WorkingFile = Trim(Directory)//Trim(InputFolder)//SourceData

Open(20,file=WorkingFile)
Read(20,*)P_S
Allocate(S_Coeff(0:P_S))
Allocate(S_CoeffO(0:P_S))
Allocate(SourceMom(0:P_S))
```

```
Do i=0,P_S
    Read(20,*)S_Coeff(i)
End Do
Close(20)

!Gather Rho Data
WorkingFile = Trim(Directory)//Trim(InputFolder)//RhoData

Open(20,file=WorkingFile)
Read(20,*)P_rho


If(P_rho==-1)then
    Sinrho = .True.
    Allocate(rho_sin(1:4))
    Do i=1,4
        Read(20,*)rho_sin(i)
    End Do
    P_rho=rho_sin(4)
    Allocate(rho_Coeff(0:P_rho))
    Allocate(rho_CoeffO(0:P_rho))
Else
    Sinrho= .False.
    Allocate(rho_Coeff(0:P_rho))
    Allocate(rho_CoeffO(0:P_rho))
    Do i=0,P_rho
        Read(20,*)rho_Coeff(i)
    End Do
End If

Close(20)

currentstep=0
```

```
    !Gather EC Data
    WorkingFile = Trim(Directory)//Trim(InputFolder)//ErrorControl

    Open(20,file=WorkingFile)

    Read(20,*)EC

    if(EC)then
        Read(20,*)del_T
        Read(20,*)Tfinal
        Read(20,*)relTolhalf
        Read(20,*)relToldoub
        Read(20,*)del_Tmin
    End if


End Subroutine Get_Problem_Data

Subroutine Source_and_Reactivity_Calc

    Use Kinds, Only: dp
    Use Variables, Only:    del_T, N_Lifetime, P_S, P_rho,  &
                            & rho_Coeff, S_Coeff, &
                            & beta_Tot, rho_bar,  &
                            & kappa_bar, dkp_coeff, Sp_coeff, &
                            & stepcount, currentstep, Sinrho, &
                            & rho_sin
    Use Sinpoly, Only: SinReactivity

    !Use Helper, Only: SimpsonsInt
    Use RPKFunctions, Only:  SolveRhoBar

    Implicit none
```

```
Character(120):: WorkingFile
Integer:: i
Real(dp):: currenttime

currenttime= Real(currentstep,dp)*del_T

If(Sinrho)then
    Call SinReactivity (rho_sin(1),rho_sin(2),rho_sin(3), 0._dp, del_T, P_rho)
End if

!Solve for rho_bar
!rho_bar=SimpsonsInt(rhoFunc,n_simpsons,0._dp,del_T)/del_T
rho_bar=SolveRhoBar(0._dp,del_T)

!Solve for kappa_bar
kappa_bar=(beta_tot-rho_bar)/N_lifetime

!Solve for d_kappa Coeff and Sp coeff
if(.not. allocated(dkp_Coeff))Allocate(dkp_Coeff(0:P_rho))
if(.not. allocated(sp_Coeff))Allocate(sp_Coeff(0:P_S))

dkp_Coeff=rho_coeff

dkp_Coeff(0)=dkp_Coeff(0)-rho_bar

Do i=0,P_rho
    dkp_Coeff(i)=dkp_Coeff(i)/N_Lifetime
End Do

sp_Coeff=S_Coeff

Do i=1,P_rho
    dkp_Coeff=dkp_Coeff*(del_T**(Real(i,dp)))
End Do
```

```
    Do i=1,P_S
        sp_Coeff=sp_Coeff*(del_T**(Real(i,dp)))
    End Do


    !Equilibrium condition for given N(0)

!        N0=1.0E8_dp
!        Write(*,*)"C1: ",beta_i(1)*N0/(lambda_i(1)*N_Lifetime)
!        Write(*,*)"C2: ",beta_i(2)*N0/(lambda_i(2)*N_Lifetime)
!        Write(*,*)"C3: ",beta_i(3)*N0/(lambda_i(3)*N_Lifetime)
!        Write(*,*)"C4: ",beta_i(4)*N0/(lambda_i(4)*N_Lifetime)
!        Write(*,*)"C5: ",beta_i(5)*N0/(lambda_i(5)*N_Lifetime)
!        Write(*,*)"C6: ",beta_i(6)*N0/(lambda_i(6)*N_Lifetime)

    !Done



End Subroutine Source_and_Reactivity_Calc



End Module User_Data
```

```
Module SolutionSteps
!Handles program major steps towards Solution
    Implicit None
    Private
    Public SolveB

    Contains

!Subroutine SourceMoments !Solves for Moments for Source Term
!    Use Kinds, Only: dp
!    Use Variables, Only: P_S, SourceMom, kappa_bar, del_T
!    Use M0Functions, Only: M0
!    Use Helper, Only: nStart
!    Implicit None
!    Integer:: StartingOrder, i, changeNum
!    Real(dp):: x, CurrentMoment
!
!    x=kappa_bar*del_T
!
!    SourceMom(0)=M0(x)
!
!    i=1
!
!    Do
!    !Forwards recurrence
!        if(i>P_S)Exit
!        if(i>x)Exit
!        SourceMom(i)= (1._dp-Real(i,dp)*SourceMom(i-1))/x
!        i=i+1
!
!    End Do
```

```
!
!    changeNum=i
!
!    if(P_S > x)then
!    !Solve for starting order of moment for backwards recurrence
!    StartingOrder = nStart(P_S,x)
!
!    !Solve for Moment at StartingOrder M_StartingOrder(x)
!
!    CurrentMoment = 1._dp/(2._dp*(Real(StartingOrder,dp)+1._dp))
!
!    Do i= StartingOrder-1, P_S+1, -1
!
!        CurrentMoment = (1._dp-x*CurrentMoment)/((Real(i,dp)+1._dp)
!
!    End Do
!
!    SourceMom(P_S) = (1._dp-x*CurrentMoment)/((Real(i,dp)+1._dp)
!
!    Do i = P_S-1, changeNum, -1
!
!        SourceMom(i) = (1._dp-x*SourceMom(i+1))/((Real(i,dp)+1._dp)
!
!    End Do
!
!    End if
!    !*****************************************
!    !Put all these goodies into M0functions to be used for any order anytime?
!    !*****************************************
!
!End Subroutine


Subroutine SolveB    !Solves for B1 and B0
```

```
Use Kinds, Only: dp
Use Variables, Only:    N0, kappa_bar, del_t, c_i0, lambda_i, sp_coeff, &
                        & B0, B1, TotalGroups,kbdt
Use RPKFunctions, Only: SFunc
Use MomentFunctions
Implicit None
Integer::i
Real(dp),Allocatable::temp1(:),temp2(:),temp3(:)

Allocate(temp1(1))
Allocate(temp2(2))
Allocate(temp3(3))

kbdt = kappa_bar*del_t
temp1(1)=kbdt

!Solve for B0

B0=N0*M(0,1,temp1)

temp2(1)=kbdt

Do i=1,TotalGroups
    temp2(2)=lambda_i(i)*del_t
    B0 = B0 + c_i0(i)*lambda_i(i)*del_t*M(0,2,temp2)
End do

Do i=0,P_S
    B0=B0 + ((sp_coeff*del_t*M(i+1,1,temp1))/Real((i+1),dp))
End Do

!Solve for B1

B1=N0*M(1,1,temp1)
```

```
    Do i=1,TotalGroups
        temp2(2)=lambda_i(i)*del_t
        B1 = B1 + c_i0(i)*lambda_i(i)*del_t*M(1,2,temp2)
    End do


    Do i=0,P_S
        B1=B1 + ((sp_coeff*del_t*M(i+2,1,temp1))/(Real((i+2),dp)*Real((i+1),dp)))
    End do

End Subroutine SolveB



!Subroutine SolveB  !Solves for B1 and B0
!    Use Kinds, Only: dp
!    Use Variables, Only:    N0, kappa_bar, del_t, c_i0,, lambda_i, S_coeff, &
!                            & b_integral, B0, B1, TotalGroups
!    Use RPKFunctions, Only: SFunc
!    Use MomentFunctions
!    Implicit None
!    Integer::i
!    Real(dp), Allocatable :: innerint(:), outerint(:), tempint(:)
!    Real(dp):: stepsize, temptime
!    Real(dp), Allocatable :: temp1(:),temp2(:),temp3(:)
!
!    !*****************************************************
!    !Solve the double integral using trapizodal rule (for B1), should this be done with exp
moment func?
!    !*****************************************************
!
!    Allocate(innerint(0:b_integral))
!    Allocate(outerint(0:b_integral))
```

```
!    Allocate(tempint(0:b_integral))
!
!    innerint(0)=0
!    outerint(0)=0
!    tempint(0)=0
!
!    stepsize = del_t/b_integral
!
!    !Need to fix the SFunc function... include source moments whoops....
!
!    Do i=1, b_integral
!        temptime = Real(i,dp)*stepsize
!        tempint(i) = SFunc(temptime)*exp(kappa_bar*temptime)
!        innerint(i)=innerint(i-1) + stepsize*(tempint(i-1)+tempint(i))/2._dp
!    End Do
!
!    Do i=1,b_integral
!        temptime = Real(i,dp)*stepsize
!        tempint(i)=innerint(i)*exp(-1._dp*kappa_bar*temptime)
!        tempint(i)=tempint(i)*(1._dp-(temptime/del_t))/del_t
!        outerint(i)=outerint(i-1) + stepsize*(tempint(i-1)+tempint(i))/2._dp
!    End Do
!
!    !Solve for B1
!
!    B1=outerint(b_integral) + N0*M(1,1,(kappa_bar*del_t))
!
!    Allocate(temp(2))
!
!    temp(1)=kappa_bar*del_t
!
!    Do i=1,TotalGroups
!        temp(2)=lambda_i(i)*del_t
!        B1=B1 + c_i0(i)*lambda_i(i)*del_t*M(1,2,temp)
```

```
!    End Do
!
!    !Solve the double integral using trapizodal rule (for B0)
!
!    Do i=1,b_integral
!        temptime = Real(i,dp)*stepsize
!        tempint(i)=innerint(i)*exp(-1._dp*kappa_bar*temptime)/del_t
!        outerint(i)=outerint(i-1) + stepsize*(tempint(i-1)+tempint(i))/2._dp
!    End Do
!
!    !Solve for B0
!
!    B0=outerint(b_integral) + N0*M(0,1,(kappa_bar*del_t))
!
!
!    Do i=1,TotalGroups
!        temp(2)=lambda_i(i)*del_t
!        B0=B0 + c_i0(i)*lambda_i(i)*del_t*M(0,2,temp)
!    End Do
!
!End Subroutine

Subroutine SolveAlphaA !Solves for alpha and a
    Use Kinds, Only: dp
    Use Variables, Only: a, alpha, del_t, B0
    Use RPKFunctions, Only: fadtFunc
    Use Helper, Only: NewtonsMet
    Implicit None
    !Note: Tol and guess are arbitraily selected, need better guess (bisect?)
    alpha = NewtonsMet(fadtFunc,1000,1._dp,.0001_dp,.01_dp)/del_t

    a = B0/A0Func(alpha*del_t)

End Subroutine
```

```
Subroutine SolveAlphaAExact
    Use Kinds, Only: dp
    Use RPKFunctions, Only: fadtFunc, dA1func, dA0func
    Use Variables, Only: del_t, P_rho, dkp_coeff, kappa_bar,TotalGroups, &
                        & beta_i, lambda_i, N_Lifetime ,a, alpha, &
                        & B0, B1
    Implicit None
    Real(dp)::current,absTol,slope,next, dA0, dA1
    Integer::maxiter, i
    !Note: Tol and guess are arbitraily selected, need better guess (bisect?)

    current=1._dp   !initial guess
    maxiter=1000    !max num of iterations
    absTol=.0001_dp !absolute Tol.

    !*********************************************************
    !is this solution wrong? check derivatives
    !perhaps above method would be more reliable
    !*********************************************************


    Do i=1,maxiter
        slope = B0*dA1func(current) - B1*dA0func(current)
        next=current-(fadtFunc(current)/slope)
        if(abs(fadtFunc(next)<absTol)alpha=next
        if(abs(fadtFunc(next)<absTol)Exit
        current=next
        if(i==maxiter)STOP "max iterations reached on newton's method"
    End Do


End Subroutine
```

```
Subroutine SolveAlphaAFalsePosition

!*******************************************
!how do I determine 2 good starting points?
!*******************************************

    Use Kinds, Only: dp
    Use RPKFunctions, Only: fadtFunc, dA1func, dA0func
    Use Variables, Only: del_t, P_rho, dkp_coeff, kappa_bar,TotalGroups, &
                        & beta_i, lambda_i, N_Lifetime ,a, alpha, &
                        & B0, B1
    Implicit None
    Real(dp)::current,absTol,pos1,pos2, slope, temp
    Integer::maxiter, i

    current=1._dp   !initial guess
    maxiter=1000    !max num of iterations
    absTol=.0001_dp !absolute Tol.

    !finding 2 positions hopefully

    if(abs(fadtFunc(1))>abs(fadtFunc(-1)))then
    pos1=1
    else
    pos1=-1
    end if

    if(fadtFunc(pos1)>0)then
        if(fadtFunc(pos1-1)<fadtFunc(pos1))then
            Do i=1, maxiter
                if (fadtFunc(pos1-i)<0)then
                    pos2=pos1-i
                    exit
                end if
```

```
        end do
    else
        Do i=1, maxiter
            if (fadtFunc(pos1+i)<0)then
                pos2=pos1+i
                exit
            end if
        end do
    end if
else
    if(fadtFunc(pos1-1)>fadtFunc(pos1))then
        Do i=1, maxiter
            if (fadtFunc(pos1-i)>0)then
                pos2=pos1-i
                exit
            end if
        end do
    else
        Do i=1, maxiter
            if (fadtFunc(pos1+i)>0)then
                pos2=pos1+i
                exit
            end if
        end do
    end if
end if

!can be rewritten to be wayyyyy more efficient

do i=1,maxiter

slope = (fadtFunc(pos2)-fadtFunc(pos1))/(pos2-pos1)
temp=((-1*fadtFunc(pos1))/slope)+pos1
```

```
    if(abs(fadtFunc(temp))<absTol)then
        alpha=temp
        exit
    end if

    if(fadtFunc(temp)>0)then
        if(fadtFunc(pos1)>0)then
            pos1=temp
        else
            pos2=temp
        end if
    else
        if(fadtFunc(pos1)<0)then
            pos1=temp
        else
            pos2=temp
        end if
    end if

    end do

End Subroutine

Subroutine SolveAlpharobust
     Use Kinds, Only: dp
    Use RPKFunctions, Only: fadtFunc
     Use Helper, Only: BisectionStarter
     Use Variables, Only: N_Lifetime, alpha, del_t
     Use Rootsolvers, Only: Secant_Bisection

    Implicit None
     Real(dp)::aa,bb !bracketing points
     Integer::errint
     Real(dp)::alphadt
```

```
      Logical::rootWorked


    !Find two starting points
      Call BisectionStarter(f=fadtFunc,a=aa,b=bb,smalliter=N_Lifetime,maxIter=100)
      !if(errint==2)STOP "max iterations during bracket finding"

      rootWorked=Secant_Bisection(x=alphadt, f=fadtFunc, a=aa, b=bb)
      if(rootWorked)then
            alpha=alphadt/del_t
      else
            STOP"Failed to find root"
      end if

End Subroutine SolveAlpharobust



Subroutine SolveA
      Use Variables, Only: a, alpha, del_t, B0
    Use RPKFunctions, Only: A0Func
      Use Kinds
      Implicit None

      a = B0/A0Func(alpha*del_t)

End Subroutine SolveA


Subroutine SolveNC !Solves for neutron and precursor amounts at dt
    Use Kinds, Only: dp
    Use MomentFunctions
    Use Variables, Only: Ndt, Ndtapprox, C_idt, C_i0, lambda_i, del_t, &
                        & a, beta_i, N_Lifetime, alpha,, N0, kappa_bar, &
```

```
                    & P_S, sp_Coeff, TotalGroups, P_rho, dkp_coeff
Implicit None
Integer::i
Real(dp)::temp
Real(dp)::temp1(1),temp2(2)


!Solve for N(dt) approximation

Ndtapprox= a*exp(alpha*del_t)

!Solve for Precursor groups

 Do i=1, TotalGroups
    C_idt(i) = c_i0(i)*exp(-1._dp*lambda(i)*del_t)
    temp1(1)=alpha*del_t+lambda_i(i)*del_t
    C_idt(i)=C_idt(i) +a*beta_i(i)*del_t*exp(alpha*del_t)*M(0,1,temp1)/N_lifetime
End Do

!Solve for N(dt)
Allocate(tempM(2))
temp1(1)=kappa_bar*del_t
Ndt=N0*exp(-1._dp*kappa_bar*del_t)

Do i=0, P_S
    Ndt=Ndt+ sp_coeff(i)*del_t*M(i,1,temp1)
End Do

Do i= 1, TotalGroups
    temp2(1)=kappa_bar*del_t-lambda_i(i)*del_t
    temp2(2)=-1._dp*alpha*del_t-lambda_i(i)*del_t
    temp=a*beta_i(i)*del_t*M(0,2,tempM)/N_lifetime
       temp1(1)=kappa_bar*del_t-lambda_i(i)*del_t
    temp=temp+c_i0(i)*M(0,1,temp1)
```

```
        Ndt=Ndt+ (temp*lambda_i(i)*del_t*exp(-1._dp*lambda_i(i)*del_t)))
    End Do

    temp1(1)=kappa_bar*del_t+alpha*del_t

    Do i=0, P_rho
        Ndt=Ndt+ a*del_t*dkp_Coeff(i)*exp(alpha*del_t)*M(i,1,temp1)
    End Do



End Subroutine SolveNC

End Module SolutionSteps
```

VIII.B.5 Finalize

```
Module Finalize
Implicit None

    Private
    Public RecordSolution, Initial_Condition_Reset

Contains

Subroutine RecordSolution
Use Variables, Only: Ndt, Ndtapprox, C_idt, TotalGroups, directory, OutputFolder, alpha, a
Implicit None
Character(120)::WorkingFile
Integer::i
```

```
WorkingFile = Trim(Directory)//Trim(OutputFolder)//'Output.txt'
Open(unit=20, file=WorkingFile)

Write(20,*)"alpha is: "
Write(20,*)alpha
Write(20,*)"a is: "
Write(20,*)a

Write(20,*)"Approximate N at dt:"
Write(20,*)Ndtapprox

Write(20,*)"N at dt:"
Write(20,*)Ndt

Write(20,*)"Precursor Groups (starting at 1):"

Do i=1,TotalGroups
    Write(20,*)C_idt(i)
End Do

End Subroutine RecordSolution

Subroutine Initial_Condition_Reset
    Use Variables, Only: Ndt,C_idt,N0,C_i0,P_rho, P_S, S_Coeff, rho_coeff,del_T,Sinrho, rho_sin
    Use Kinds, Only: dp
    Use Helper, Only: choose
    Implicit None
    Real(dp)::newrho(0:P_rho),newS(0:P_S)
    Integer::i,j


    N0=Ndt
    C_i0=C_idt
```

```
    newrho=0._dp
    newS=0._dp

    !shift rho and source (needs del_T of previous)
    if(Sinrho)then
        rho_sin(2) = rho_sin(2)+ rho_sin(3)*del_T
    else
        Do i=0,P_rho
            Do j=i,P_rho
                newrho(i)=newrho(i)+rho_coeff(j)*(del_T**(j-i))*Real(choose(j,i),dp)
            End do
        End do
    end if

    Do i=0,P_S
        Do j=i,P_S
            newS(i)=newS(i)+S_coeff(j)*(del_T**(j-i))*Real(choose(j,i),dp)
        End do
    End do

    rho_coeff=newrho
    S_coeff=newS



End Subroutine Initial_Condition_Reset

End Module Finalize
```

```
Module M0_Functions
    Use Kinds, Only: dp
    Implicit None

    Interface M0
        Module Procedure M0Scalar
        Module Procedure M0Vector
    End Interface M0


    Private
    Public:: M0, RunM0Tests, M

    Real(dp), Parameter:: Large = 39._dp
    Real(dp), Parameter:: Small = 0.6931471805599453_dp ! Log[2]

Contains


Function M0Scalar(x) Result(f)
    ! Calculates the order 0 rank 1
    ! exponential moment function M0(x)
    ! using a variety of algorithms such that
    ! error is always negligible:
    ! less than or about 1 bit in Real(dp)
    Use Kinds, Only: dp
    Implicit None
    Real(dp):: f
    Real(dp), Intent(In):: x
    Real(dp):: ax, m
    Integer:: j
```

```
    ax = Abs(x)
    If (ax > Large) then
        !Exp[-39.] < 1.16E-17
        m = 1._dp / x
    Else if (ax < Small) then
        ! Maclaurin Series, equivalent to
        ! Backward recurrence from M15(x)
        ! to M0(x) where
        ! M15(x) = 0.6127 (+/-0.012)
        ! for 0 <= x <= Log[2.]
        ! so that |error in M0| < 4.E-18
        m = 0.06127_dp
        Do j = 15, 1, -1
            m = (1._dp - ax * m) / Real(j,dp)
        End do
    Else
        ! Loss of precision <= 0.4 digit (about 1 bit)
        ! for x >= Log[2.]
        m = (1._dp - Exp(-x)) / x
    End if
    If (x < 0._dp) then
        f = m * Exp(ax)
    Else
        f = m
    End if
End Function M0Scalar

!Probably move these next couple functions since they're not M0

!Function M(order,k,x) result (f)
!    Use Kinds, Only: dp
!    Use Sorters, Only: Sort, ReverseOrder
!    Use Helper, Only: nStart
!    Implicit None
```

```
!    Real(dp):: f
!    Real(dp),intent(in):: x(1:k)
!    Integer,intent(in)::k
!    Integer,intent(in)::order
!    Real(dp)::y(1:k)
!    Integer::Startingorder, i
!    Real(dp)::CurrentMoment
!
!
!    if(order==0 .and. k>1)then
!        f=M0Vector(x,k)
!        return
!    else if(order==0 .and. k==1)then
!        f=M0(x(1))
!        return
!    else if(k<1)then
!        Stop "Error in calling M0(x,k): k < 1 "
!    else if(k>1 .and. order>1)then
!    !Need some help for k>1 order>1 case
!    !somehow have to reduce k to 1... not sure if I can just use decay chain
!    !conditioning??
!    y=x
!    Call Sort(y, k)
!    Call ReverseOrder(y, k)
!    f = MVectorHelper(order,y, k)
!
!
!    else !k==1 order>1
!
!        if(order>x(1))then !backwards recurrence
!        Startingorder= nstart(order,x(1))
!        CurrentMoment = 1._dp/(2._dp*(Real(StartingOrder,dp)+1._dp))
!
!        Do i= StartingOrder-1, order, -1
```

```
!
!            CurrentMoment = (1._dp-x(1)*CurrentMoment)/((Real(i,dp)+1._dp)
!
!        End Do
!
!            f=CurrentMoment
!
!
!        else !forward recurrence
!
!        CurrentMoment = M0(x(1))
!
!        Do i=1,order
!            CurrentMoment =  (1._dp-Real(i,dp)*CurrentMoment)/x(1)
!        End do
!
!        end if
!
!    end if
!
!End Function
!
!Recursive Function MVectorHelper(order, x, k) Result(f)
!    Use Kinds, Only: dp
!    Use Helper, Only: nStart
!    Implicit None
!    Real(dp):: f
!    Real(dp), Intent(In):: x(1:k)   ! argument
!    Integer, Intent(In) :: k        ! rank
!    Integer,intent(in)::order
!    Integer::Startingorder, i
!    Real(dp)::CurrentMoment
!
!    if(order==0 .and. k>1)then
```

```
!        f=M0Vector(x,k)
!        return
!    else if(order==0 .and. k==1)then
!        f=M0(x(1))
!        return
!    else if(k<1)then
!        Stop "Error in calling M0(x,k): k < 1 "
!    else if(k>1 .and. order>1)then
!
!    !Will break for small x
!    f=(MVectorhelper(order,x(2:k),k-1)-(order*MVectorhelper(order-1,x,k)))/x(1)
!
!    else !k==1 order>1
!
!        if(order>x(1))then !backwards recurrence
!        Startingorder= nstart(order,x(1))
!        CurrentMoment = 1._dp/(2._dp*(Real(StartingOrder,dp)+1._dp))
!
!        Do i= StartingOrder-1, order, -1
!
!            CurrentMoment = (1._dp-x(1)*CurrentMoment)/((Real(i,dp)+1._dp)
!
!        End Do
!
!            f=CurrentMoment
!
!
!        else !forward recurrence
!
!        CurrentMoment = M0(x(1))
!
!        Do i=1,order
!            CurrentMoment =  (1._dp-Real(i,dp)*CurrentMoment)/x(1)
!        End do
```

```
!
!        end if
!
!    end if
!
!
!End Function MVectorHelper

Function M(order, k, x) Result (f)
     ! Calculates the order "order" rank "k"
     ! exponential moment function
     Use Sorters, Only: Sort, ReverseOrder
     Use Kinds, Only: dp
     Use Helper, Only: nStart
     Implicit None
     Real(dp)::f
     Real(dp), Intent(In)::x(1:k)
     Integer, Intent(In) :: k
     Integer, Intent(In) :: order
    Real(dp):: y(1:k)
     Integer::Startingorder, i
     Real(dp)::CurrentMoment

     if (order==0) then
          f = M0Vector(x,k)
          Return
     else if (k<1)
          Stop "Error in calling moment (k < 1)"
     else if (k==1 .AND. order>x(1)) then !backwards recurrence
          Startingorder= nstart(order,x(1))
       CurrentMoment = 1._dp/(2._dp*(Real(StartingOrder,dp)+1._dp))

       Do i= StartingOrder-1, order, -1
```

```
            CurrentMoment = (1._dp-x(1)*CurrentMoment)/((Real(i,dp)+1._dp)

        End Do

        f=CurrentMoment

      else if (k==1 .AND. order<=x(1)) then
            CurrentMoment = M0(x(1))

            Do i=1,order
             CurrentMoment =  (1._dp-Real(i,dp)*CurrentMoment)/x(1)
        End do

            f= CurrentMoment

      else !k>1
      y = x
      Call Sort(y,k)
      Call ReverseOrder(y,k) !decreasing order, x(1) is largest
      f= MHelp(order, k, y)

      end if
End Function M

Recursive Function MHelp(order, x, k) result (f)
      Use Kinds, Only: dp
    Implicit None
    Real(dp):: f
    Real(dp), Intent(In):: x(1:k)   ! argument
    Integer, Intent(In) :: k        ! rank
      Integer, Intent(In) :: order
    Real(dp), Parameter:: closeTol = 0.5_dp
    Real(dp):: smallest, largest, kminus1
    smallest = x(k)
```

```
largest = x(1)

 if (order==0) then
       f = M0Vector(x,k)
       Return
 else if (k<1)
       Stop "Error in calling moment (k < 1)"
 else if (k==1 .AND. order>x(1)) then !backwards recurrence
       Startingorder= nstart(order,x(1))
    CurrentMoment = 1._dp/(2._dp*(Real(StartingOrder,dp)+1._dp))

    Do i= StartingOrder-1, order, -1

        CurrentMoment = (1._dp-x(1)*CurrentMoment)/((Real(i,dp)+1._dp)

    End Do

    f=CurrentMoment

 else if (k==1 .AND. order<=x(1)) then
       CurrentMoment = M0(x(1))

       Do i=1,order
        CurrentMoment =  (1._dp-Real(i,dp)*CurrentMoment)/x(1)
    End do

       f= CurrentMoment

 else if (largest>small) then !k>1, n>0, there is one somewhat large positive argument
       CurrentMoment= Real(order,dp)*MHelp(order-1,x,k)
       CurrentMoment = MHelp(order, x(2:k),k-1)-CurrentMoment
       f= CurrentMoment/x(1)

 else if (abs(smallest)>small !k>1,n>0, there is one somewhat large negative argument
```

```
            CurrentMoment= Real(order,dp)*MHelp(order-1,x,k)
            kminus1 = k - 1
            CurrentMoment = MHelp(order, x(1:kminus1),k-1)-CurrentMoment
            f=CurrentMoment/x(k)

      else !k>1, n>0 all arguments are between -smallest and smallest (.693)
            f = MMaclaurin(order,x,k)
      end if

End function MHelp

Function MMaclaurin(order,x,k) Result(f)
      Use Kinds, Only: dp
      Implicit None
      Real(dp):: f
    Real(dp), Intent(In):: x(1:k)   ! argument
    Integer, Intent(In) :: k        ! rank
      Integer, Intent(In) :: order
    Real(dp), Parameter:: relTol = 1.E-17_dp
      Real(dp):: d, p(1:k), s, t
    Integer:: j,kordersum, orderplus1

      kordersum= k + order
      orderplus1 = order + 1

      d = 1._dp
    p = 1._dp
    Do j = orderplus1, kordersum
        d = d * Real(j,dp)
    End do

      s = 1._dp / d
    j = kordersum
```

```
      Do
            j = j + 1
            d = -d * Real(j,dp)
            p = p * x
            t = RunningSum(p) / d
            s = s + t
            If (Abs(t) <= relTol * s) Exit
    End do
    f = s
End Function MMaclaurin


Function M0Vector(x, k) Result(f)
    ! Calculates the order 0 rank k
    !   exponential moment function M0(x(1:k))
    ! M0 is an orderless function.
    !   To facilitate minimizing loss of precision,
    !   this function sorts a copy of the
    !   argument vector into increasing order,
    !   reverses it to decreasing order,
    !   and calls a recursive helper
    !   that evaluates M0 without further sorting.
    Use Sorters, Only: Sort, ReverseOrder
    Use Kinds, Only: dp
    Implicit None
    Real(dp):: f
    Real(dp), Intent(In):: x(1:k)   ! argument
    Integer, Intent(In) :: k        ! rank
    Real(dp):: y(1:k)
    If (k==1) then
        ! call M directly to avoid needless overhead
        f = M0(x(1))
        Return
    Else if (k < 1) then
```

```
        Stop "Error in calling M0(x,k): k < 1 "
    Else
        ! Use a copy  of x so that x can be intent(In)
        ! i.e., eliminate side effect of changing x
        y = x
        Call Sort(y, k)
        Call ReverseOrder(y, k)
        f = M0VectorHelper(y, k)
    End if
End Function M0Vector

Recursive Function M0VectorHelper(x, k) Result(f)
    Use Kinds, Only: dp
    Implicit None
    Real(dp):: f
    Real(dp), Intent(In):: x(1:k)   ! argument
    Integer, Intent(In) :: k        ! rank
    Real(dp), Parameter:: closeTol = 0.5_dp
    Real(dp):: smallest, largest
    smallest = x(k)
    largest = x(1)
    If (k==1) then
        ! Special case to which larger cases are reduced
        f = m0(x(1))
    Else if (smallest >= Large) then
        ! all the arguments > large
        f = 1._dp / Product(x)
    Else if (All(Abs(x) < Small)) then
        f = M0Maclaurin(x, k)
    Else if (Abs(largest-smallest) <= &
           & closeTol * (Abs(smallest) + Abs(largest))/2._dp) then
        ! All close together
        ! Use decay chain function recurrence
        f = (M0VectorHelper(x(2:k),k-1) - &
```

```
            & Exp(-x(k))*M0VectorHelper(x(1:k-1)-x(k),k-1)) / x(1)
! @@@ add check to use x(1) or X(K) BASED ON LARGER IN MAGNITUDE
   Else
        ! use recurrence on rank
        f = (M0VectorHelper(x(2:k),k-1) - &
                M0VectorHelper(x(1:k-1),k-1))&
           & / (x(1) - x(k))
    End if
End Function M0VectorHelper

Function M0Maclaurin(x, k) Result(f)
    Use Kinds, Only: dp
    Implicit None
    Real(dp):: f
    Real(dp), Intent(In):: x(1:k)   ! argument
    Integer, Intent(In) :: k        ! rank
    Real(dp), Parameter:: relTol = 1.E-17_dp
    Real(dp):: d, p(1:k), s, t
    Integer:: j
    d = 1._dp
    p = 1._dp
    Do j = 2, k
        d = d * Real(j,dp)
    End do
    s = 1._dp / d
    j = k
    Do
        j = j + 1
        d = -d / Real(j,dp)
        p = p * x
        t = RunningSum(p) / d
        s = s + t
        If (Abs(t) <= relTol * s) Exit
    End do
```

```
    f = s
End Function M0Maclaurin

Function RunningSum(x)
    ! The result value is the same as
    !   that of the intrinsic Sum function,
    !   but RunningSum differs by its side effect:
    !   after returning, x(j) is the sum of
    !   the original x(1:j) for j=1,k
    Use Kinds, Only: dp
    Implicit None
    Real(dp):: RunningSum
    Real(dp), Intent(InOut):: x(1:)
    Integer :: k
    Integer:: j
    k = Size(x,1)
    Do j = 2, k
        x(j) = x(j) + x(j-1)
    End do
    RunningSum = x(k)
End Function RunningSum


Subroutine RunM0Tests
    Use Kinds, Only: dp
    Implicit None
    Integer :: order
    Real(dp):: x, f, factor
    Integer :: rank
    Character(len=1):: choice
    Do
        Print *, "Test of M0 function."
        Write (*, "(1x,a)",Advance='no') "Enter Rank:  k = "
        Read (*,*) rank
```

```
        Write (*, "(1x,a)",Advance='no') "Enter base x:  x = "
        Read (*,*) x
        Write (*, "(1x,a)",Advance='no') "Enter factor for growth of x entries:  factor = "
        Read (*,*) factor
        Print *
        f = TestM0(x, rank, factor)
        Write (*, "(1x,a)",Advance='no') "Run M0 test again? (y/n): "
        Read(*,*) choice
        Do
            If (choice=='y' .or. choice=='n' .or. choice=='Y' .or. choice =='N') Exit
            Write (*, "(1x,a)",Advance='no') 'Please enter either "y" or "n": '
            Read(*,*) choice
        End do
        If (choice=="n" .or. choice=="N") Exit
    End do
End Subroutine RunM0Tests

Function TestM0(xBase, rank, factor) Result(f)
    Use Kinds, Only: dp
    Implicit None
    Real(dp):: f
    Real(dp), Intent(In):: xBase
    Integer,  Intent(In):: rank
    Real(dp), Intent(In):: factor
    Real(dp):: x(1:rank)
    Integer:: j
    x(1) = xBase
    Do j = 2, rank
        x(j) = x(j-1) * factor
    End do
    f = M0(x, rank)
    Print *, "M0(", x, ") ="
    Print *, f
End Function TestM0
```

```
End Module M0_Functions
```

VIII.B.7 Kinds

```
Module Kinds
    Implicit None
    Integer, Parameter:: sp = Selected_Real_Kind(p=6)
    Integer, Parameter:: dp = Selected_Real_Kind(p=15)
End Module Kinds
```

VIII.B.8 Helper

```
Module Helper
Implicit None

Contains

Function SimpsonsInt(f,n,a,b) Result(answer) !function, n intervals (even), low bound, up bound
Use kinds, Only:dp
Implicit None
Integer, Intent(in)::n
Real(dp), Intent(in)::a,b

Interface
function f(x) result(ans)
```

```
Use kinds, only: dp
implicit none
real(dp), intent(in)::x
real(dp)::ans
end function f
end interface

Real(dp):: answer, h
Integer::j

h= (b-a)/Real(n,dp)

answer=f(a)+f(b)

Do j=1,((n/2)-1)
    answer=answer+ 2._dp*f(a+(2._dp*Real(j,dp))*h)
End Do

Do j=1, n/2
    answer=answer +4._dp*f(a+(2._dp*Real(j,dp)-1._dp)*h)
End Do

answer=answer*(h/3._dp)

End Function



Function NewtonsMet(f,maxiter,initialguess,absTol,dx) Result(next)
Use Kinds, Only:dp
Implicit None

Interface
function f(x) result(ans)
```

```
Use kinds, only: dp
implicit none
real(dp), intent(in)::x
real(dp)::ans
end function f
end interface

Integer, Intent(in)::maxiter
Real(dp), Intent(in)::initialguess
Real(dp), Intent(in)::absTol
Real(dp), Intent(in)::dx  !half size of derivative measurement
Real(dp)::current, next, slope
Integer::i

current=initialguess

Do i=1,maxiter
    slope= (f(current+dx)-f(current-dx))/(2._dp*dx)
    next=current - (f(current)/slope)
    if(abs(f(next))<absTol)Exit
    current=next
    if(i==maxiter)STOP "max iterations reached on newton's method"
End do



End Function

Function nStart(nMax,x) Result(nS)
Use Kinds, Only: dp
Implicit None
Integer:: nS
Integer, Intent(In):: nMax ! Max order of M[n,x] to be evaluated
Real(dp), Intent(In):: x
```

```
Real(dp):: xPower, FactorialRatio, Tol, denom
Real(dp), Parameter:: RelTol = 1.E-15_dp
If (x < 0._dp .or. nMax < 1 .or. x > nMax) &
& STOP "nStart: illegal argument"
Tol = RelTol / (Real(nMax+1,dp)+x)
nS = nMax
xPower = x
FactorialRatio = 1._dp / Real(nS+1,dp)
Do
nS = nS + 1
xPower = xPower * x
FactorialRatio = FactorialRatio / Real(nS+1,dp)
denom = 2._dp * (Real(nS+1,dp)+x) * (Real(nS+2,dp)+x)
If (xPower * FactorialRatio / denom <= Tol) Exit
End do
End Function nStart

Subroutine BisectionStarter(f,p,b,smalliter,maxIter, err)
!Finds two starting points that contain the root (a and b) for func f
     Use Kinds, Only: dp
    Implicit None
     Interface
        Function f(x)
            Use Kinds, Only: dp
            Implicit None
            Real(dp):: f
            Real(dp), Intent(In):: x
        End Function f
    End Interface
     Real(dp), Intent(Out):: a, b
     Real(dp), Intent(In), Optional:: smalliter, maxIter
     Integer, Intent(Out), Optional:: err
     Real(dp)::itersmall, fa, fb, iterMax
     Integer::iter
```

```
  If (Present(smalliter)) then
    itersmall = smalliter
Else
    itersmall = 1.E-4_dp
End if

  If (Present(maxIter)) then
    iterMax = maxIter
Else
    iterMax = 50
End if

  If (Present(err)) err = 0

  a=0._dp
  fa=f(0)
  if(fa==0)then
       a=-1._dp*itersmall
       fa=f(a)
  end if
  if(isNAN(fa))then
       a=-1._dp*itersmall/3._dp
       fa=f(a)
  end if


  iter=0

  if(fa>0)then !positive f at a
       Do
            b=a+itersmall*((2)**iter)
            fb=f(b)
            if (fb<0)Return
```

```
            if(iter>iterMax)return
      End do
      if(iter>iterMax)then
            iter=0
            Do
                  b=a-itersmall*((2)**iter)
                  fb=f(b)
                  if (fb<0)Return
                  if(iter>iterMax)return
            End do
      end if
else !negative f at a
      Do
            b=a+itersmall*((2)**iter)
            fb=f(b)
            if (fb>0)Return
            if(iter>iterMax)return
      End do
      if(iter>iterMax)then
            iter=0
            Do
                  b=a-itersmall*((2)**iter)
                  fb=f(b)
                  if (fb>0)Return
                  if(iter>iterMax)return
            End do
      end if

end if

if(iter>iterMax)then
      If (Present(err)) err = 2 !max iter reached
end if
```

```
End Subroutine BisectionStarter


End Module
```

VIII.B.9 Root solvers

```
Module Rootsolvers

    Implicit None
    Private
    Public:: Bisection , NewtonSolver, Converged

Contains

Function Bisection(x, f, xL, xR, fRoot, absTol, relTol, maxIter, err) Result(foundRoot)
    ! Finds x such that f(x) = fRoot using bisection
    Use Kinds, Only: dp
    Implicit None
    Logical:: foundRoot
    Real(dp), Intent(Out):: x
    Interface
        Function f(x)
            Use Kinds, Only: dp
            Implicit None
            Real(dp):: f
            Real(dp), Intent(In):: x
        End Function f
    End Interface
    Real(dp), Intent(InOut):: xL, xR   ! initial bounds on x
```

```
Real(dp), Intent(In), Optional:: fRoot
Real(dp), Intent(In), Optional:: absTol, relTol
Integer, Intent(In), Optional:: maxIter
Integer, Intent(Out), Optional:: err
Real(dp):: fAtRoot
Real(dp):: xC, fC, fL, fR
Logical:: HighLeft, HighRight, HighCenter
Real(dp):: aTol, rTol
Integer:: iter, iterMax
If (Present(fRoot)) then
    fAtRoot = fRoot
Else
    fAtRoot = 0._dp
End if
If (Present(absTol)) then
    aTol = absTol
Else
    aTol = 0._dp
End if
If (Present(relTol)) then
    rTol = relTol
Else
    rTol = 1.E-8_dp
End if
If (Present(maxIter)) then
    iterMax = maxIter
Else
    iterMax = 50
End if
If (Present(err)) err = 0   ! presume success
FoundRoot = .True.
fL = f(xL)
If (fL == fAtRoot) then
    x = xL
```

```
    Return
Else
    HighLeft = (fL > fAtRoot)
End if
fR = f(xR)
If (fR == fAtRoot) then
    x = xR
    Return
Else
    HighRight = (fR > fAtRoot)
End if
If (HighLeft == HighRight) then
    If (Present(err)) err = 1 ! Need initial brackets such that fRoot is between fL and fR
    foundRoot = .False.
    Return
End if
iter = 0
Do
    iter = iter + 1
    xC = (xL+xR) / 2._dp
    If (Converged(xL, xR, aTol, rTol)) then
        x = xC
        Return
    End if
    fC = f(xC)
    If (IsNaN(fC)) then
        If (Present(err)) err = -1 ! xC is out of function's defined domain
        foundRoot = .False.
        Return
    End if
    If (fC == fAtRoot) then
        x = xC
        Return
    Else
```

```
            HighCenter = (fC > fAtRoot)
        End if
        If (HighLeft == HighCenter) then
            xL = xC
        Else
            xR = xC
        End if
        If (iter > iterMax) Then
            If (Present(err)) err = 2 ! too many iterations
            foundRoot = .False.
        End if
    End do
End Function Bisection


Function NewtonSolver(x, f, dfdx, fRoot, absTol, relTol, maxIter, err) Result(foundRoot)
    ! Finds x such that f(x) = fRoot using bisection
    Use Kinds, Only: dp
    Implicit None
    Logical:: foundRoot
    Real(dp), Intent(InOut):: x
    Interface
        Function f(x)
            Use Kinds, Only: dp
            Implicit None
            Real(dp):: f
            Real(dp), Intent(In):: x
        End Function f
        Function dfdx(x)
            Use Kinds, Only: dp
            Implicit None
            Real(dp):: dfdx
            Real(dp), Intent(In):: x
        End Function dfdx
```

```
End Interface
Real(dp), Intent(In), Optional:: fRoot
Real(dp), Intent(In), Optional:: absTol, relTol
Integer, Intent(In), Optional:: maxIter
Integer, Intent(Out), Optional:: err
Real(dp):: fAtRoot
Real(dp):: xOld, y, dydx
Real(dp):: aTol, rTol
Integer:: iter, iterMax
If (Present(fRoot)) then
    fAtRoot = fRoot
Else
    fAtRoot = 0._dp
End if
If (Present(absTol)) then
    aTol = absTol
Else
    aTol = 0._dp
End if
If (Present(relTol)) then
    rTol = relTol
Else
    rTol = 1.E-8_dp
End if
If (Present(maxIter)) then
    iterMax = maxIter
Else
    iterMax = 50
End if
If (Present(err)) err = 0   ! presumes success
foundRoot = .True.
iter = 0
Do
    iter = iter + 1
```

```
        xOld = x
        y = f(x)
        If (IsNaN(y)) then
            If (Present(err)) err = -1 ! x is out of function's defined domain
            foundRoot = .False.
            Return
        End if
        dydx = dfdx(x)
        If (IsNaN(dydx)) then
            If (Present(err)) err = -2 ! x is out of derivative function's defined domain
            foundRoot = .False.
            Return
        End if
        x = x - (y - fAtRoot) / dydx
        If (Converged(x, xOld, aTol, rTol)) Return
        If (iter > iterMax) Then
            If (Present(err)) err = 2 ! too many iterations
            foundRoot = .False.
            Return
        End if
    End do
End Function NewtonSolver


Function Secant_Simple(f, a, b, absTol, relTol, maxIter) Result(x)
    ! Finds x such that f(x) = 0 using Secant method
    ! Secant uses the slope of the chord to approximate f'(x) in Newton's method
    ! Thus it has simlar failure modes.
    ! The starting points must be "close enough" to the root,
    !   which depends on the shape of f.
    ! This function is used as follows:     var = Secant_Simple(...)
    Use Kinds, Only: dp
    Implicit None
    Real(dp):: x
```

```
Interface
    Function f(x)
        Use Kinds, Only: dp
        Implicit None
        Real(dp):: f
        Real(dp), Intent(In):: x
    End Function f
End Interface
Real(dp), Intent(In):: a, b     ! search interval is [a,b]
Real(dp), Intent(In), Optional:: absTol, relTol
Integer, Intent(In), Optional:: maxIter
Real(dp):: x0, x1, x2, f0, f1, f2
Integer:: iterMax
Real(dp):: aTol, rTol
Integer:: iter
If (Present(absTol)) then
    aTol = absTol
Else
    aTol = 0._dp
End if
If (Present(relTol)) then
    rTol = relTol
Else
    rTol = 1.E-10_dp
End if
If (Present(maxIter)) then
    iterMax = maxIter
Else
    iterMax = 50
End if
x0 = a; f0 = f(x0)
x1 = b; f1 = f(x1)
iter = 0
Do
```

```
        iter = iter + 1
        x2 = (f1*x0 - f0*x1)/(f1 - f0)
        If (Converged(x1, x2, aTol, rTol)) then
            x = x2
            Return
        End if
        If (iter >= iterMax) then
            Print *, "Secant_Simple failed to converge"
            Print *, "x0 = ", x0
            Print *, "x1 = ", x1
            Print *, "x2 = ", x2
            Print *, "f0 = ", f0
            Print *, "f1 = ", f1
            Print *, "f2 = ", f2
            Stop
        End if
        f2 = f(x2)
        x0 = x1; f0 = f1
        x1 = x2; f1 = f2
    End do
End Function Secant_Simple


Function Secant_Bisection(x, f, a, b, fRoot, absTol, relTol, maxIter, err) Result(foundRoot)

    ! Finds x such that f(x) = fRoot using Secant

    ! Maintains bisection bounds and uses bisection when secant jumps outside the bounds.
    ! This requires that the interval (a,b) is a bounding interval with
    !       f(a) <= fRoot <= f(b)   or   f(b) <= fRoot <= f(a)

    ! Tightens the bounds at each iteration.

    ! This function returns a logical value to indicate whether the root was found.
```

```
! Therefore it is used as follows:

!   ... Code to find bounds as above around the desired root ...
!   Do
!       If (Secant_Bisection(var, ...) .and. root is desired root) Exit
!       ... Code to find better bounds (to get correct root)
!   End do

! If you know there is only one root, (as when f(x) is continuous and monotonic everywhere)
!   use it as follows:

!   ... Code to find bounds as above around the desired root ...
!   If (.Not. Secant_Bisection(var, ...) then
!       Print *, error message and info for troubleshooting
!       Stop
!   End if

! Convergence is guaranteed if
!   f(x) is continuous in [a,b],
!   f(x) is monotonic in [a,b],
!   a <= root <= b   OR  a <= root <= b,
!   0 < Min(Abs(f'(x))) in [a,b],
!   and enough iterations are allowed.

Use Kinds, Only: dp
Implicit None
Logical:: foundRoot
Real(dp), Intent(Out):: x
Interface
    Function f(x)
        Use Kinds, Only: dp
        Implicit None
        Real(dp):: f
        Real(dp), Intent(In):: x
```

```
    End Function f
End Interface
Real(dp), Intent(In):: a, b     ! search interval is [a,b]
Real(dp), Intent(In), Optional:: fRoot
Real(dp), Intent(In), Optional:: absTol, relTol
Integer, Intent(In), Optional:: maxIter
Integer, Intent(Out), Optional:: err
Real(dp):: fAtRoot              ! Finds x such that f(x) = fAtRoot, default is fAtRoot = 0._dp
Real(dp):: xL, xR               ! Refined search interval
Real(dp):: fL, fR
Real(dp):: x0, x1, x2           ! Secant search points
Real(dp):: f0, f1, f2
Logical:: HighLeft, HighRight, HighCenter
Real(dp):: aTol, rTol
Integer:: iter, iterMax
If (Present(fRoot)) then
    fAtRoot = fRoot
Else
    fAtRoot = 0._dp
End if
If (Present(absTol)) then
    aTol = absTol
Else
    aTol = 0._dp
End if
If (Present(relTol)) then
    rTol = relTol
Else
    rTol = 1.E-10_dp
End if
If (Present(maxIter)) then
    iterMax = maxIter
Else
    iterMax = 50
```

```
End if
If (Present(err)) err = 0   ! presume success
FoundRoot = .True.
xL = Min(a,b)
xR = Max(a,b)
fL = f(xL)
If (Converged(fL, fAtRoot, aTol, rTol)) then
    x = xL
    Return
Else
    HighLeft = (fL > fAtRoot)
End if
fR = f(xR)
If (Converged(fR, fAtRoot, aTol, rTol)) then
    x = xR
    Return
Else
    HighRight = (fR > fAtRoot)
End if
If (HighLeft == HighRight) then
    If (Present(err)) err = 1 ! Need initial bracket does not guarantee root
    foundRoot = .False.
    Return
End if
iter = 0
x0 = xL; f0 = fL
x1 = xR; f1 = fR
Do
    iter = iter + 1
    x2 = (f1*x0 - f0*x1)/(f1 - f0)
    If (x2 <= xL .or. x2 >= xR) then
        x2 = (xL+xR) / 2._dp
        x0 = xL; f0 = fL
        x1 = xR; f1 = fR
```

```
        End if
        If (Converged(x1, x2, aTol, rTol)) then
            x = x2
            Return
        End if
        f2 = f(x2)
        If (Converged(f2, fAtRoot, aTol, rTol)) then
            x = x2
            Return
        End if
        If (IsNaN(f2)) then
            If (Present(err)) err = -1 ! xC is out of function's defined domain
            foundRoot = .False.
            Return
        End if
        If (iter > iterMax) Then
            If (Present(err)) err = 2 ! too many iterations
            foundRoot = .False.
            Return
        End if
        HighCenter = (f2 > fAtRoot)
        If (HighLeft == HighCenter) then
            xL = x2
        Else
            xR = x2
        End if
        x0 = x1; f0 = f1
        x1 = x2; f1 = f2
    End do
End Function Secant_Bisection



Function Converged(x, y, absTol, relTol) Result (OK)
```

```
    Use Kinds, Only: dp
    Implicit None
    Logical:: OK
    Real(dp), Intent(In):: x, y         ! Values to compare
    Real(dp), Intent(In):: absTol, relTol   ! tolerances
    Real(dp):: avgAbs, absDif
    avgAbs = (Abs(x) + Abs(y)) / 2._dp
    absDif = Abs(x-y)
    OK = ( absDif <= absTol) .or. (absDif <= relTol * avgAbs)
End Function Converged


End Module Rootsolvers
```

VIII.B.10 Reactor Kinetics Functions

```
Module RPKFunctions
    Implicit None
    Private
    Public RhoFunc, SFunc

Contains

Function RhoFunc(t) Result(rho) !function that returns reactivity given time
    Use Kinds, Only: dp
    Use Variables, Only:    P_rho, rho_coeff
    Implicit None
    Real(dp), Intent(In):: t
    Real(dp)::rho
    Integer::i
```

```
    rho=0._dp
    Do i=0, P_rho
        rho=rho+rho_coeff(i)*(t**i)
    End Do

End Function RhoFunc

Function SFunc(t) Result (S) !function that returns source given time
    Use Kinds, Only: dp
    Use Variables, Only:    P_S, S_coeff
    Implicit None
    Real(dp), Intent(In):: t
    Real(dp)::S
    Integer::i

    !Needs to use source moments to be correct... fix this

    S=0._dp
    Do i=0, P_S
        S=S+S_coeff(i)*(t**i)
    End Do

End Function SFunc

Function delRhoFunc(t) Result (delRho)
!function that returns difference in current reactivity vs average reactivity given time

    Use Kinds, Only: dp
    Use Variables, Only: rho_bar
    Implicit None
    Real, Intent(in):: t
    Real(dp):: delRho
```

```
    delRho=rhoFunc(t)-rho_bar
End Function delRhoFunc

Function kappaFunc(t) Result (kappa)
!function that returns kappa given time

    Use Kinds, Only: dp
    Use Variables, Only: beta_tot, N_Lifetime
    Implicit None
    Real, Intent(in):: t
    Real(dp):: kappa

    kappa=(beta_tot-rhoFunc(t))/N_Lifetime
End Function kappaFunc

Function delKappaFunc(t) Result (delKappa)
!function that returns the difference in current kappa and kappa bar

    Use Kinds, Only: dp
    Use Variables, Only: kappa_bar
    Implicit None
    Real, Intent(in)::t
    Real(dp):: delKappa

    delKappa=kappa_bar-kappaFunc(t)

End Function delKappaFunc

Function A1Func(adt) Result (A1)
!function that returns A1 given alpha*del_t

    Use Kinds, Only: dp
    Use Variables, Only: del_t,P_rho,dkp_coeff, kappa_bar,TotalGroups, &
                        & beta_i, lambda_i, N_Lifetime
```

```
    Use MomentFunctions
    Real, Intent)(in) :: adt
    Integer :: i
    Real(dp) :: temp3(3), temp1(1)
    Real(dp) :: A1

    temp3(1)=-1._dp*kappa_bar*del_t - adt
    temp3(2)=-1._dp*adt
    temp3(3)=-1._dp*adt
     temp1(1)=-1._dp*adt

    A1=M(1,1,temp1)

    Do i=0,P_rho
        A1=A1- del_t*dkp_coeff(i)*M(i,3,temp3)
    End do

    temp3(1)=kappa_bar*del_t

    Do i=1, TotalGroups
        temp3(2)=lambda(i)*del_t
        A1=A1-beta_i(i)*lambda_i(i)*del_t*del_t*M(1,3,temp3)/N_Lifetime
    End Do

End Function A1Func

Function A0Func(adt) Result (A0)
!function that returns A0 given alpha*del_t

    Use Kinds, Only: dp
    Use Variables, Only: del_t,P_rho,dkp_coeff, kappa_bar,TotalGroups, &
                        & beta_i, lambda_i, N_Lifetime
    Use MomentFunctions
    Real(dp), Intent)(in) :: adt
```

```
    Integer :: i
    Real(dp) :: temp3(3), temp2(2), temp1(1)
    Real(dp) :: A0


    temp2(1)=-1._dp*kappa_bar*del_t - adt
    temp2(2)=-1._dp*adt
     temp1(1)=-1._dp*adt

    A0=M(0,1,temp1)

    Do i=0,P_rho
        A0=A0- del_t*dkp_coeff(i)*M(i,2,temp2)
    End do

    temp3(1)=kappa_bar*del_t
    temp3(3)=-1._dp*adt

    Do i=1, TotalGroups
        temp3(2)=lambda(i)*del_t
        A0=A0-beta_i(i)*lambda_i(i)*del_t*del_t*M(0,3,temp3)/N_Lifetime
    End Do

End Function

Function fadtFunc(adt) Result(f) !Special Equation used in root solving for finding alpha
    Use Kinds, Only: dp
    Use Variables, Only: B1, B0
    Real(dp), Intent(in)::adt
    Real(dp) :: f

    f= B0*A1Func(adt) - B1*A0(adt)

End Function A0func
```

```
Function dA1func(adt) Result(dA1)
    Use Kinds, Only: dp
    Use Variables, Only: del_t,P_rho,dkp_coeff, kappa_bar,TotalGroups, &
                        & beta_i, lambda_i, N_Lifetime
    Use MomentFunctions
    Real, Intent)(in) :: adt
    Integer :: i
    Real(dp), Allocatable :: temp2(:),temp4(:)
    Real(dp) :: dA1

    Allocate(temp2(2))
    Allocate(temp4(4))

    temp2(1)=-1._dp*adt
    temp2(2)=-1._dp*adt

    temp4(1)=-1._dp*kappa_bar*del_t - adt
    temp4(2)=-1._dp*adt
    temp4(3)=-1._dp*adt
    temp4(4)=-1._dp*adt

    dA1=M(1,2,temp2)

    Do i=0,P_rho
        dA1=dA1+ 6._dp*del_t*dkp_coeff(i)*M(i,4,temp4)
    End do

    temp4(1)=kappa_bar*del_t
    temp4(3)=-1._dp*adt
    temp4(4)=-1._dp*adt

    Do i=1, TotalGroups
        temp4(2)=lambda(i)*del_t
```

```
        dA1=dA1+beta_i(i)*lambda_i(i)*del_t*del_t*M(1,4,temp4)/N_Lifetime
    End Do

End Function dA1func

Function dA0func(adt) Result(dA0)
    Use Kinds, Only: dp
    Use Variables, Only: del_t,P_rho,dkp_coeff, kappa_bar,TotalGroups, &
                        & beta_i, lambda_i, N_Lifetime
    Use MomentFunctions
    Real(dp), Intent)(in) :: adt
    Integer :: i
    Real(dp), Allocatable :: temp2(:), temp3(:),temp4(:)
    Real(dp) :: dA0

    Allocate(temp2(2))
    Allocate(temp3(3))
    Allocate(temp4(4))

    temp3(1)=-1._dp*kappa_bar*del_t - adt
    temp3(2)=-1._dp*adt
    temp3(3)=-1._dp*adt

    temp2(1)=-1._dp*adt
    temp2(2)=-1._dp*adt

    dA0=-1._dp*M(0,2,temp2)

    Do i=0,P_rho
        dA0=dA0- 2._dp*del_t*dkp_coeff(i)*M(i,3,temp3)
    End do

    temp4(1)=kappa_bar*del_t
    temp4(3)=-1._dp*adt
```

```
    temp4(4)=-1._dp*adt

    Do i=1, TotalGroups
        temp4(2)=lambda(i)*del_t
        dA0=dA0+beta_i(i)*lambda_i(i)*del_t*del_t*M(0,4,temp4)/N_Lifetime
    End Do
End Function dA0func

End Module
```

VIII.B.11 Sin Poly

```
Module SinPoly
Implicit None

Contains

    Subroutine SinReactivity (aa0, bb0, cc0, ti, dt, order)

        Use Kinds, Only: dp
        Use Helper, Only: choose
        Use Variables, Only: rho_coeff
        Implicit None
        Real(dp), Intent(In) :: aa0, bb0, cc0, ti, dt ! 5 initial inputs
        ! a0 * Sin(b0 + c0 * t) for t = ti to ti + dt
        ! We want to fit a polynomial to this using moments of the Legendre polynomials
        Integer,Intent(In) :: order ! order of solution of interest
        Real(dp) :: p(0:4,0:4) !Legendre Polynomial Coefficients: p(n,coefficient order)
        Real(dp) :: aa,bb,cc !sin constants after variable change to u (between -1 and 1)
        Real(dp) :: cip(0:4) !inner product of sin and Legendre Poly
        Real(dp) :: fu(0:4,0:4) ! Polynomial representation variable u f(n,coefficient order)
```

```
Real(dp) :: ft(0:4) !Polynomial representation variable t ff(coefficient order)
Real(dp) :: aat, bbt !transformation constants
Integer :: i, j


!Set up table of Legendre Polynomial Coefficients
p=0._dp

p(0,0) = 1._dp

p(1,1) = 1._dp

p(2,0) = -.5_dp
p(2,2) = 1.5_dp

p(3,1) = -1.5_dp
p(3,3) = 2.5_dp

p(4,0) = .125_dp
p(4,2) = 3.75_dp
p(4,4) = 4.375_dp

!change function for u = -1 to 1

aa = aa0
bb = bb0 + (cc0 * dt / 2._dp) + cc0 * ti
cc = cc0 * dt / 2._dp

!Solve inner product constants

cip(0) = aa*sin(bb)*sin(cc)/cc

cip(1) = cos(bb)*sin(cc)/(cc**2._dp)
cip(1) = cip(1) - cos(bb)*cos(cc)/cc
```

```
cip(1) = aa*cip(1)

cip(2) = 3._dp*cos(cc)*sin(bb)/(cc**2._dp)
cip(2) = cip(2) - 3._dp*sin(bb)*sin(cc)/(cc**3._dp)
cip(2) = cip(2) + sin(bb)*sin(cc)/cc
cip(2) = aa*cip(2)

cip(3) = 15._dp*cos(bb)*cos(cc)/(cc**3._dp)
cip(3) = cip(3) - cos(bb)*cos(cc)/cc
cip(3) = cip(3) - 15._dp*cos(bb)*sin(cc)/(cc**4._dp)
cip(3) = cip(3) + 6._dp*cos(bb)*sin(cc)/(cc**2._dp)
cip(3) = aa*cip(3)

cip(4) = -105._dp*cos(cc)*sin(bb)/(cc**4._dp)
cip(4) = cip(4) + 10._dp*cos(cc)*sin(bb)/(cc**2._dp)
cip(4) = cip(4) + 105._dp*sin(bb)*sin(cc)/(cc**5._dp)
cip(4) = cip(4) - 45._dp*sin(bb)*sin(cc)/(cc**3._dp)
cip(4) = cip(4) + sin(bb)*sin(cc)/cc
cip(4) = aa*cip(4)

!Set up f, the polynomial equivelant with respect to u

fu=0._dp

Do i=0,4
    fu(i,0) = cip(0)*p(0,0)
End Do

Do i = 1,4
    fu(i,1) = fu(i,1) + cip(1)*3._dp*p(1,1)
End Do

Do i = 2,4
    fu(i,0) = fu(i,0) + cip(2)*5._dp*p(2,0)
```

```
            fu(i,2) = fu(i,2) + cip(2)*5._dp*p(2,2)
        End Do

        Do i=3,4
            fu(i,1) = fu(i,1) + cip(3)*7._dp*p(3,1)
            fu(i,3) = fu(i,3) + cip(3)*7._dp*p(3,3)
        End Do

            fu(4,0) = fu(4,0) + cip(4)*9._dp*p(4,0)
            fu(4,2) = fu(4,2) + cip(4)*9._dp*p(4,2)
            fu(4,4) = fu(4,4) + cip(4)*9._dp*p(4,4)

        !Transform fu to ft (back into correct time domain

        aat = -1._dp - (2._dp*ti/dt)
        bbt = 2._dp/dt

        ft=0._dp

        Do i=0,order
            Do j=i,order
                ft(i)=ft(i)+fu(order,j)*(aat**(j-i))*Real(choose(j,i),dp)*(bbt**i)
            End do
        End do

        Do i = 0, order
            rho_coeff(i)=ft(i)
        End Do

    End Subroutine SinReactivity

End Module SinPoly
```

```
Module Sorters
    Implicit None

    Interface Sort
        Module Procedure BubbleSort_dp
        Module Procedure BubbleSort_Integer
    End Interface Sort

    Interface ReverseOrder
        Module Procedure ReverseOrder_dp
        Module Procedure ReverseOrder_Integer
    End Interface ReverseOrder

    Private
    Public:: Sort, ReverseOrder
    Public:: Test_Sort_and_ReverseOrder_dp
    Public:: Test_Sort_and_ReverseOrder_Integer

Contains

Subroutine Test_Sort_and_ReverseOrder_dp
    Use Kinds, Only: dp
    Implicit None
    Integer, Parameter:: n = 4
    Real(dp):: x(1:n)
    x = Real( (/ 5, 3, 4, 2 /), dp)
    Print *, "X before sorting = ", x
    Call Sort(x, n)
    Print *, "X after sorting = ", x
    Call ReverseOrder(x, n)
    Print *, "X after sorting and then reversing order = ", x
```

```
End Subroutine Test_Sort_and_ReverseOrder_dp

Subroutine BubbleSort_dp(a, n)
    Use Kinds, Only: dp
    Implicit None
    Integer, Intent(In):: n
    Real(dp), Intent(InOut):: a(1:n)
    Integer:: j,k
    Do j = 1, n-1
        Do k = n, j+1, -1
            If ( a(k) < a(k-1) ) Call Swap_dp(a(k), a(k-1))
        End do
    End do
End Subroutine BubbleSort_dp

Subroutine ReverseOrder_dp(a, n)
    Use Kinds, Only: dp
    Implicit None
    Integer, Intent(In):: n
    Real(dp), Intent(InOut):: a(1:n)
    Integer:: j
    Do j = 1, n/2
        Call Swap_dp(a(j), a(n+1-j))
    End do
End Subroutine ReverseOrder_dp

Subroutine Swap_dp(x, y)
    Use Kinds, Only: dp
    Implicit None
    Real(dp), Intent(InOut):: x, y
    Real(dp):: SwapHolder
    SwapHolder = x
    x = y
    y = SwapHolder
```

```
End Subroutine Swap_dp

Subroutine Test_Sort_and_ReverseOrder_Integer
    Use Kinds, Only: dp
    Implicit None
    Integer, Parameter:: n = 4
    Integer:: x(1:n)
    x = (/ 5, 3, 4, 2 /)
    Print *, "X before sorting = ", x
    Call Sort(x, n)
    Print *, "X after sorting = ", x
    Call ReverseOrder(x, n)
    Print *, "X after sorting and then reversing order = ", x
End Subroutine Test_Sort_and_ReverseOrder_Integer

Subroutine BubbleSort_Integer(a, n)
    Implicit None
    Integer, Intent(In):: n
    Integer, Intent(InOut):: a(1:n)
    Integer:: j,k
    Do j = 1, n-1
        Do k = n, j+1, -1
            If ( a(k) < a(k-1) ) Call Swap_Integer(a(k),a(k-1))
        End do
    End do
End Subroutine BubbleSort_Integer

Subroutine ReverseOrder_Integer(a, n)
    Implicit None
    Integer, Intent(In):: n
    Integer, Intent(InOut):: a(1:n)
    Integer:: j
    Do j = 1, n/2
        Call Swap_Integer(a(j), a(n+1-j))
```

```
    End do
End Subroutine ReverseOrder_Integer

Subroutine Swap_Integer(x, y)
    Implicit None
    Integer, Intent(InOut):: x, y
    Integer:: SwapHolder
    SwapHolder = x
    x = y
    y = SwapHolder
End Subroutine Swap_Integer

End Module Sorters
```

VIII.C. Mathematica Worksheet

```
(* ClearAll["Global`*"] *)
N0 = 1*10^8;
delT =50;

TotalGroups = 6;

betai={.00021,.00142,.00127,.00257,.00075,.00027};
betatot=Sum[k,{k,betai}];

NLifetime = 12.7*betatot + 3*(1-betatot)*10^-5;

lambdai = {.0126,.0301,.112,.301,1.14,3.01};
```

```
Ci0constant=1*10^4;
Ci0={20213583.5233573,57215857.6807323,13752455.9328556,10355264.3830621,797904.612764104
,108790.715308435};
Cidt=Range[TotalGroups];

PS=2;
SCoeff:={SCoeff0,SCoeff1,SCoeff2}; (*length PS+1*)

SCoeff0=0;
SCoeff1=0;
SCoeff2=0;

Prho=2;
rhoCoeff:={rhoCoeff0,rhoCoeff1,rhoCoeff2}

rhoCoeff0=0.001;
rhoCoeff1=0.000001;
rhoCoeff2=0;

rhofunc=0;
Do[rhofunc=rhofunc+rhoCoeff[[n+1]]*t^n,{n,0,Prho,1}];

rhofunc=0;
rhofunc=.001Sin[6.28t];

Sfunc=0;
Do[Sfunc=Sfunc+SCoeff[[n+1]]*t^n,{n,0,PS,1}];

s=NDSolve[{n'[t]          □((rhofunc
betatot)*n[t]/NLifetime)+Sfunc+lambdai[[1]]*C1[t]+lambdai[[2]]*C2[t]+lambdai[[3]]*C3[t]+l
ambdai[[4]]*C4[t]+lambdai[[5]]*C5[t]+lambdai[[6]]*C6[t],C1'[t]==(betai[[1]]*n[t]/NLifetim
```

```
e)-lambdai[[1]]*C1[t],C2'[t]==(betai[[2]]*n[t]/NLifetime)-
lambdai[[2]]*C2[t],C3'[t]==(betai[[3]]*n[t]/NLifetime)-
lambdai[[3]]*C3[t],C4'[t]==(betai[[4]]*n[t]/NLifetime)-
lambdai[[4]]*C4[t],C5'[t]==(betai[[5]]*n[t]/NLifetime)-
lambdai[[5]]*C5[t],C6'[t]==(betai[[6]]*n[t]/NLifetime)-lambdai[[6]]*C6[t], n[0]==N0,
C1[0]==Ci0[[1]],C2[0]==Ci0[[2]],C3[0]==Ci0[[3]],C4[0]==Ci0[[4]],C5[0]==Ci0[[5]],C6[0]==Ci
0[[6]]},{n,C1,C2,C3,C4,C5,C6},{t,0,delT},WorkingPrecision □30, Accurac
l    □15];
Plot[Evaluate[n[t]/.s],{t,0,delT},PlotRange    □All]

{n[delT]/.s,{
   C1[delT]/.s,
   C2[delT]/.s,
   C3[delT]/.s,
   C4[delT]/.s,
   C5[delT]/.s,
   C6[delT]/.s}}//FullForm
{1.03552643830621`*^7}//FullForm
108787.76821347115`
n[delT]
n[49.5]/.s//FullForm
C6[50.]/.s//FullForm
```

IX. BIBLIOGRAPHY

[1] Mathews, Kirk. *Exponential Moment Methods.*

[2] Shultis, Kenneth J., Faw, Richard E. *Fundamentals of Nuclear Science and Engineering.* Marcel Dekker, Inc., 2nd Edition, 2002.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE (DD–MM–YYYY)	2. REPORT TYPE	3. DATES COVERED *(From — To)*
21-03-2013	Master's Thesis	Oct 2011 – Mar 2013

4. TITLE AND SUBTITLE
Solving Point-Reactor Kinetics Equations Using Exponential Moment Methods

5a. CONTRACT NUMBER
5b. GRANT NUMBER
5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Thelen, Paul

5d. PROJECT NUMBER
5e. TASK NUMBER
5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/ENY)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT-ENP-13-M-34

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Intentionally left blank.

10. SPONSOR/MONITOR'S ACRONYM(S)
11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

A robust method of solving the reactor point kinetic equations was designed using exponential moment methods. Although the method requires a relatively large number of calculations to complete, the accuracy ensured by each individual step calculation allows larger time steps to be used. The algorithm designed was verified to converge to the correct value as step size was reduced. Additionally, the algorithm can take steps much larger than the average neutron lifetime while maintaining some precision. An error control scheme was designed based on changes observed in the results as a function of time step size. The error control adaptively approaches optimal step sizes within a factor of two for given tolerances. When used in conjunction with our algorithm, most cases show large mitigation of computational cost.

15. SUBJECT TERMS
Point-Reactor Kinetics Equations, Exponential Moment Methods, Error Control

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE	UU	209	Dr. Kirk Mathews, ENP
U	U	U			19b. TELEPHONE NUMBER (Include Area Code) (937)255-3636, ext 4508

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18